Instructor's Manual for Sipser's

# Introduction to the Theory of COMPUTATION

CHING LAW
EDMOND KAYI LEE
ZULFIKAR RAMZAN

Massachusetts Institute of Technology
Laboratory for Computer Science

MICHAEL SIPSER

Massachusetts Institute of Technology

BROOKS/COLE PUBLISHING COMPANY
ITP• An International Thomson Publishing Company

Pacific Grove • Albany • Belmont • Boston • Cincinnati • Johannesburg • London • Madrid
Melbourne • Mexico City • New York • Scottsdale • Singapore • Tokyo • Toronto

COPYRIGHT© 1999 by Brooks/Cole Publishing Company
A division of International Thomson Publishing Inc.
ITP The ITP logo is a registered trademark used herein under license.

*For more information, contact:*

BROOKS/COLE PUBLISHING COMPANY
511 Forest Lodge Road
Pacific Grove, CA 93950
USA

International Thomson Publishing Europe
Berkshire House 168-173
High Holborn
London WC1V 7AA
England

Thomas Nelson Australia
102 Dodds Street
South Melbourne, 3205
Victoria, Australia

Nelson Canada
1120 Birchmount Road
Scarborough, Ontario
Canada M1K 5G4

International Thomson Editores
Seneca 53
Col. Polanco
11560 México, D. F., México

International Thomson Publishing GmbH
Königswinterer Strasse 418
53227 Bonn
Germany

International Thomson Publishing Asia
60 Albert Street
#15-01 Albert Complex
Singapore 189969

International Thomson Publishing Japan
Palaceside Building, 5F
1-1-1 Hitotsubashi
Chiyoda-ku, Tokyo 100-0003
Japan

All rights reserved. Instructors of classes using *Introduction to the Theory of Computation* by Sipser as a textbook may reproduce material from this publication for classroom use. Otherwise, the text of this publication may not be reproduced, stored in a retrieval system, or transcribed, in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher, Brooks/Cole Publishing Company, Pacific Grove, California 93950.

Printed in the United States of America.

5 4 3 2 1

ISBN 0-534-37462-X

# Table of Contents


Chapter 0 solutions . . . . . 1
Chapter 1 solutions . . . . . 1
Chapter 2 solutions . . . . . 11
Chapter 3 solutions . . . . . 24
Chapter 4 solutions . . . . . 32
Chapter 5 solutions . . . . . 37
Chapter 6 solutions . . . . . 44
Chapter 7 solutions . . . . . 44

# Preface

This instructor's manual is designed to accompany the textbook, *Introduction to the Theory of Computation*, by Michael Sipser, PWS Publishers, 1997. It contains solutions to almost all of the exercises and problems appearing in Chapters 1–5 and 7 along with a handful of solutions to problems in other chapters. Most of the omitted solutions in the early chapters require figures, and producing these required more work that we were able to put into this manual at this point. The next edition will be more complete.

This manual is available only to instructors and only in hardcopy form. Requests for copies should be directed to: 606–282–4629, fax 606–647–5020.

Thanks to Jonathan Feldman for sending in corrections to a draft of this manual.
An errata site for this instructor's manual maintained at

`http://www-math.mit.edu/~sipser/itocsm-errs1.1.html` .

The email address for correspondence regarding the textbook and the instructor's manual is `sipserbook@math.mit.edu` .

# Chapter 0

**0.12** Here is a sketch of the solution. Make two piles, A and B, of nodes; initially empty. Then, starting with the entire graph, add each remaining node to A if its degree is greater than 1/2 of all remaining nodes and to B otherwise, then discard all nodes to which it isn't (is) connected if it was added to A (B). Continue until nothing is left.

# Chapter 1

**1.1**
- **a.** The start state of $M_1$ is $q_1$.
- **b.** The set of accept states of $M_1$ is $\{q_2\}$.
- **c.** The start state of $M_2$ is $q_1$.
- **d.** The set of accept states of $M_2$ is $\{q_1, q_4\}$.
- **e.** On input aabb $M_1$ goes through the state sequence $q_1, q_2, q_3, q_1, q_1$.
- **f.** $M_1$ does not accept *aabb*.
- **g.** $M_2$ does accept $\varepsilon$.

**1.2**
- **a.** $M_1 = (\{q_1, q_2, q_3\}, \{\mathtt{a}, \mathtt{b}\}, \delta_1, q_1, \{q_2\})$.
  The transition function $\delta_1$ is

| | a | b |
|---|---|---|
| $q_1$ | $q_2$ | $q_1$ |
| $q_2$ | $q_3$ | $q_3$ |
| $q_3$ | $q_2$ | $q_1$ |

- **b.** $M_2 = (\{q_1, q_2, q_3, q_4\}, \{\mathtt{a}, \mathtt{b}\}, \delta_2, q_1, \{q_1, q_4\})$.
  The transition function $\delta_2$ is

| | a | b |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_4$ |
| $q_3$ | $q_2$ | $q_1$ |
| $q_4$ | $q_3$ | $q_4$ |

**1.9** Let $N = (Q, \Sigma, \delta, q_0, F)$ be any NFA. We build an NFA $N'$ with only a single accept state that accepts the same language as $N$. Informally, $N'$ looks exactly like $N$ except it has $\varepsilon$-transitions from states corresponding to accept states of $N$ to a new special accept state, $q_{\text{accept}}$. Finally, the state $q_{\text{accept}}$has no transitions coming out of it. More formally, $N' = (Q \cup \{q_{\text{accept}}\}, \Sigma, \delta', q_0, \{q_{\text{accept}}\})$. Where for any $q \in Q$ and $a \in \Sigma$:

$$\delta'(q,a) = \begin{cases} \delta(q,a) & \text{if } a \neq \varepsilon \text{ or } q \notin F \\ \delta(q,a) \cup \{q_{\text{accept}}\} & \text{if } a = \varepsilon \text{ and } q \in F \end{cases}$$

and $\delta'(q_{\text{accept}}, a) = \emptyset$ for $a \in \Sigma_\varepsilon$.

**1.10** **a.** Let $M'$ be the DFA $M$ with the accept and non-accept states swapped. We will show that $M'$ recognizes the complement of $B$, where $B$ is the language recognized by $M$. Suppose $M'$ accepts $x$. If we run $M'$ on $x$ we end in an accept state of $M'$. Because $M$ and $M'$ have swapped accept/non-accept states, if we run $M$ on $x$, we would end in a non-accept state. Therefore, $x \notin B$. Similarly, if $x$ is not accepted by $M'$, then it would be accepted by $M$. So $M'$ accepts exactly those strings not accepted by $M$. Therefore, $M'$ recognizes the complement of $B$.

Since $B$ could be any arbitrary regular language and our construction shows how to build an automaton to recognize its complement, it follows that the complement of any regular language is also regular. Therefore, the class of regular languages is closed under complement.

**b.** Consider the NFA in exercise 1.12(a). The string $a$ is accepted by this automaton. If we swap the accept and reject states, the string $a$ is still accepted. This shows that swapping the accept and non-accept states of an NFA doesn't necessarily yield a new NFA recognizing the complement of the original one. The class of languages recognized by NFAs is, however, closed under complement. This follows from the fact that the class of languages recognized by NFAs is precisely the class of languages recognized by DFAs which we know is closed under complement from part (a).

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

**a.** $1\Sigma^*0$
**b.** $\Sigma^*1\Sigma^*1\Sigma^*1\Sigma^*$
**c.** $\Sigma^*0101\Sigma^*$
**d.** $\Sigma\Sigma0\Sigma^*$
**e.** $(0 \cup 1\Sigma)(\Sigma\Sigma)^*$
**f.** $(0 \cup (10)^*)^*1^*$
**g.** $(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)$
**h.** $\Sigma^*0\Sigma^* \cup 1111\Sigma^* \cup 1 \cup \varepsilon$
**i.** $(1\Sigma)^*(1 \cup \varepsilon)$
**j.** $0^*(100 \cup 010 \cup 001)0^*$
**k.** $\varepsilon \cup 0$
**l.** $(1^*01^*01^*)^* \cup 0^*10^*10^*$
**m.** $\emptyset$
**n.** $\Sigma\Sigma^*$

**1.15** **a.** ab,$\varepsilon$;ba,aba
**b.** ab,abab;$\varepsilon$,aabb
**c.** $\varepsilon$,aa;ab,aabb
**d.** $\varepsilon$,aaa;aa,b
**e.** aba,aabbaa;$\varepsilon$,abbb
**f.** aba,bab;$\varepsilon$,ababab

g. b,ab;$\varepsilon$,bb
h. ba,bba;b,$\varepsilon$

**1.16** In both parts we first add a new start state and a new accept state. Several solutions are possible, depending on the order states are removed.

a. Here we remove state 1 then state 2 and we obtain
$\mathtt{a}^*\mathtt{b}(\mathtt{a} \cup \mathtt{ba}^*\mathtt{b})^*$

b. Here we remove states 1, 2, then 3 and we obtain
$\varepsilon \cup ((\mathtt{a} \cup \mathtt{b})\mathtt{a}^*\mathtt{b}((\mathtt{b} \cup \mathtt{a}(\mathtt{a} \cup \mathtt{b}))\mathtt{a}^*\mathtt{b})^*(\varepsilon \cup \mathtt{a}))$

**1.17** a. $A_1 = \{0^n1^n2^n \mid n \geq 0\}$. Assume that $A_1$ is regular. Let $p$ be the pumping length given by the pumping lemma. Choose $s$ to be the string $0^p1^p2^p$. Because $s$ is a member of $A_1$ and s has length more than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz$ is in $A_1$.

i) The string $y$ consists only of 0s, only of 1s or only of 2s. In these cases the string $xyyz$ will not have equal number of 0s, 1s and 2s, leading to a contradiction.

ii) The string $y$ consists of more than one kind of symbol. In this case $xyyz$ will have the 0s, 1s or 2s out of order. Hence it is not a member of $A_1$, which is a contradiction.

Therefore, $A_1$ is not regular.

b. $A_2 = \{ww \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$. Assume that $A_2$ is regular. Let $p$ be the pumping length given by the pumping lemma. Choose $s$ to be the string $\mathtt{a}^p\mathtt{b}\mathtt{a}^p\mathtt{b}$. The pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz$ is in $A_2$.

i) The string $y$ consists of a's only. In both cases that $y$ is left or right of the middle b, $xyyz$ would not be in the form of $ww$.

ii) The string $y$ is $\mathtt{a}^p\mathtt{b}$. The condition $|xy| \leq p$ would be violated.

iii) The string $y$ contains the center b and $y \neq \mathtt{a}^p\mathtt{b}$. Therefore, $xyyz$ would not be in the form of $ww$.

Since all cases cause contradictions, $A_2$ is not regular.

c. $A_3 = \{\mathtt{a}^{2^n} \mid n \geq 0\}$. Assume that $A_3$ is regular. Let $p$ be the pumping length given by the pumping lemma. Choose $s$ to be the string $\mathtt{a}^{2^p}$. The pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz$ is in $A_3$.
The shortest string in $A_3$ which is longer than $\mathtt{a}^{2^p}$ is $s' = \mathtt{a}^{2^{p+1}} = \mathtt{a}^{2\cdot 2^p}$. Thus the length of $s'$ is double the length of $s$. The only way that $xyyz$ could be in $A_3$ is by letting $y$ be $\mathtt{a}^{2^p}$. However, that would violate the pumping lemma condition that $|xy| \leq p$. Therefore, $A_3$ is not regular.

**1.18** The error is that $s = 0^p1^p$ *can* be pumped. Let $s = xyz$, where $x = 0$, $y = 0$ and $z = 0^{p-2}1^p$. The conditions are satisfied because

i) for any $i \geq 0$, $xy^iz = 00^i0^{p-2}1^p$ is in $0^*1^*$.
ii) $|y| = 1 > 0$, and
iii) $|xy| = 2 \leq p$.

**1.19**
a. $q_1, q_1, q_1, q_1$; 000.
b. $q_1, q_2, q_2, q_2$; 111.
c. $q_1, q_1, q_2, q_1, q_2$; 0101.
d. $q_1, q_3$; 1.
e. $q_1, q_3, q_2, q_3, q_2$; 1111.
f. $q_1, q_3, q_2, q_1, q_3, q_2, q_1$; 110110.
g. $q_1$; $\epsilon$.

**1.20** A *finite state transducer* is a 5-tuple $(Q, \Sigma, \Gamma, \delta, q_0)$, where

i) $Q$ is a finite set called the *states*,
ii) $\Sigma$ is a finite set called the *alphabet*,
iii) $\Gamma$ is a finite set called the *output alphabet*,
iv) $\delta\colon Q \times \Sigma \longrightarrow Q \times \Gamma$ is the *transition function*,
v) $q_0 \in Q$ is the *start state*.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a *finite state transducer*, $w = w_1w_2\cdots w_n$ be a string over $\Sigma$, and $v = v_1v_2\cdots v_n$ be a string over the $\Gamma$. Then $M$ *outputs* $v$ if a sequence of states $r_0, r_1, \ldots, r_n$ exists in Q with the following two conditions:

i) $r_0 = q_o$
ii) $\delta(r_i, w_{i+1}) = (r_{i+1}, v_{i+1})$ for $i = 0, \ldots, n-1$.

**1.21**
a. $T_1 = (Q, \Sigma, \Gamma, \delta, q_1)$, where
i) $Q = \{q_1, q_2\}$,
ii) $\Sigma = \{0, 1, 2\}$,
iii) $\Gamma = \{0, 1\}$,
iv) $\delta$ is described as

| | 0 | 1 | 2 |
|---|---|---|---|
| $q_1$ | $(q_1,0)$ | $(q_1,0)$ | $(q_2,1)$ |
| $q_2$ | $(q_1,0)$ | $(q_2,1)$ | $(q_2,1)$ |

v) $q_1$ is the start state.

b. $T_2 = (Q, \Sigma, \Gamma, \delta, q_1)$, where
i) $Q = \{q_1, q_2, q_3\}$,
ii) $\Sigma = \{a, b\}$,
iii) $\Gamma = \{0, 1\}$,
iv) $\delta$ is described as

| | a | b |
|---|---|---|
| $q_1$ | $(q_2,1)$ | $(q_3,1)$ |
| $q_2$ | $(q_3,1)$ | $(q_1,0)$ |
| $q_3$ | $(q_1,0)$ | $(q_2,1)$ |

v) $q_1$ is the start state.

**1.23** a. Assume $L = \{0^n1^m0^n \mid m,n \geq 0\}$ is regular. Let $p$ be the pumping length given by the pumping lemma. The string $s = 0^p10^p \in L$, and $|s| \geq p$. Thus the pumping lemma implies that $s$ can be divided as $xyz$ with $x = 0^a, y = 0^b, z = 0^c10^p$, where $b \geq 1$ and $a+b+c = p$. However, the string $s' = xy^0z = 0^{a+c}10^p \notin L$, since $a+c < p$. That contradicts the pumping lemma.

b. If $L$, the complement of $C = \{0^n1^n \mid n \geq 0\}$ is regular, then $C$ itself is regular, using the result of Exercise 1.10a. However, we have shown in Example 1.38 that $C$ is non-regular. So $L$ cannot be regular.

c. Assume $L = \{0^m1^n \mid m \neq n\}$ is regular. Let $p$ be the pumping length given by the pumping lemma. Set $m = p$ and $n = p+p!$. The string $s = 0^p1^{p+p!} \in L$, and $|s| \geq p$. Thus the pumping lemma implies that $s$ can be divided as $xyz$ with $x = 0^a, y = 0^b, z = 0^c1^{p+p!}$, where $b \geq 1$ and $a+b+c = p$. However, the string $s' = xy^{\frac{p!}{b}+1}z = 0^{a+p!+b+c}1^{p+p!} = 0^{p+p!}1^{p+p!} \notin L$. That contradicts the pumping lemma.

Alternative solution: The nonregular language $\{0^n1^n \mid n \geq 0\} = C \cap 0^*1^*$. Hence $C$ cannot be regular because the regular languages are closed under intersection.

d. Assume $C = \{w \mid w \in \{0,1\}^* \text{ is a palindrome}\}$ is regular. Let $p$ be the pumping length given by the pumping lemma. The string $s = 0^p10^p \in C$ and $|s| \geq p$. Follow the argument as in part (a). Hence $C$ isn't regular, so neither is its complement.

**1.24** For any regular language $A$, let $M_1$ be the DFA recognizing it. We need to find a DFA that recognizes $A^R$. Since any NFA can be converted to an equivalent DFA, it suffices to find an NFA $M_2$ that recognizes $A^R$.

We keep all the states in $M_1$ and reverse the direction of all the arrows in $M_1$. We set the accept state of $M_2$ to be the start state in $M_1$. Also, we introduce a new state $q_0$ as the start state for $M_2$ which goes to every accept state in $M_1$ by an $\epsilon$-transition.

For every string $s$ accepted by $M_1$, the path that $s$ follows in $M_1$ starts at the start state and stops at one of the accept states, say, $q_{\text{end}}$. If we input $s^R$ to $M_2$, it starts at $q_0$ and forks into different paths starting at each of the accept states in $M_1$. The computation can reach the accept state in $M_2$ by following the same path as its reversed counterpart in $M_1$ but in the opposite direction.

Similarly, for every string $w$ accepted by $M_2$, there exists a path $P$ starting at $q_0$ going through $q_{\text{end}}$ as its next step and stopping at the unique accept

state. Converting the NFA $M_2$ back to its counterpart $M_1$ and reversing the input string, we can get from the start state to $q_{\text{end}}$ in $M_1$ by following the same path in the opposite direction.

Therefore, $M_2$ recognizes $A^R$ if $M_1$ recognizes $A$.

**1.27** The idea is that we start by comparing the most significant bit of the two rows. If the bit in the top row is bigger, we know that the string is in the language. The string does not belong to the language if the bit in the top row is smaller. If the bits on both rows are the same, we move on to the next most significant bit until a difference is found. We implement this idea with a DFA having states $q_0$, $q_1$, and $q_2$. State $q_0$ indicates the result is not yet determined. States $q_1$ and $q_2$ indicate the top row is known to be larger, or smaller, respectively. We start with $q_0$. If the top bit in the input string is bigger, it goes to $q_1$, the only accept state, and stays there till the end of the input string. If the top bit in the input string is smaller, it goes to $q_2$ and stays there till the end of the input string. Otherwise, it stays in state $q_0$.

**1.28** Assume language $E$ is regular. Use the pumping lemma to a get a pumping length $p$ satisfying the conditions of the pumping lemma. Set $s = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^p \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p$. Obviously, $s \in E$ and $|s| \geq p$. Thus, the pumping lemma implies that the string $s$ can be written as $xyz$ with $x = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^a, y = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^b, z = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^c \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p$, where $b \geq 1$ and $a + b + c = p$. However, the string $s' = xy^0z = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^{a+c} \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p \notin E$, since $a+c < p$. That contradicts the pumping lemma. Thus $E$ is not regular.

**1.29** For each $n \geq 1$, we build a DFA with the $n$ states $q_0, q_1, \ldots, q_{n-1}$ to count the number of consecutive a's modulo $n$ read so far. For each character a that is input, the counter increments by 1 and jumps to the next state in $M$. It accepts the string if and only if the machine stops at $q_0$. That means the length of the string consists of all a's and its length is a multiple of $n$.

More formally, the set of states of $M$ is $Q = \{q_0, q_1, \ldots, q_{n-1}\}$. The state $q_0$ is the start state and the only accept state. Define the transition function as: $\delta(q_i, \mathtt{a}) = q_j$ where $j = i + 1 \bmod n$.

**1.30** By simulating binary division, we create a DFA $M$ with $n$ states that recognizes $C_n$. $M$ has $n$ states which keep track of the $n$ possible remainders of the division process. The start state is the only accept state and corresponds to remainder 0.

The input string is fed into $M$ starting from the most significant bit. For each input bit, $M$ doubles the remainder that its current state records, and then adds the input bit. Its new state is the sum modulo $n$. We double the remainder because that corresponds to the left shift of the computed remainder in the long division algorithm.

If an input string ends at the accept state (corresponding to remainder 0), the binary number has no remainder on division by $n$ and is therefore a member of $C_n$.

The formal definition of $M$ is $(\{q_0, \ldots, q_{n-1}\}, \{0,1\}, \delta, q_0, \{q_0\})$. For each $q_i \in Q$ and $b \in \{0,1\}$, define $\delta(q_i, b) = q_j$ where $j = (2i + b) \bmod n$.

**1.32** a. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing $A$, where $A$ is some regular language. We construct $M' = (Q', \Sigma, \delta, q_0', F')$ recognizing $NOPREFIX(A)$ as follows:

i) $Q' = Q$

ii) For $r \in Q'$ and $a \in \Sigma$ define $\delta'(r, a) = \begin{cases} \delta(r, a) & R \notin F \\ \emptyset & R \in F. \end{cases}$

iii) $q_0' = q_0$

iv) $F' = F$

b. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing $A$, where $A$ is some regular language. We construct $M' = (Q', \Sigma, \delta, q_0', F')$ recognizing $NOEXTEND(A)$ as follows:

i) $Q' = Q$

ii) $\delta' = \delta$

iii) $q_0' = q_0$

iv) $F' = \{q |$ there is no path from $q$ to an accept state$\}$.

**1.33** Assume to the contrary that some FST $T$ outputs $w^{\mathcal{R}}$ on input $w$. Consider the input strings 00 and 01. On input 00, $T$ must output 00, and on input 01, $T$ must output 10. That is impossible, because in both cases the first input bit is a 0, but in one case the first output bit is a 0 and in the other case the first output is a 1, and the first output bit depends only on the first input bit. Hence no such FST can exist.

**1.34** To show that $\equiv_L$ is an equivalence relation we show it is reflexive, symmetric, and transitive. It is reflexive because no string can distinguish $x$ from itself and hence $x \equiv_L x$ for every $x$. It is symmetric because $x$ is distinguishable from $y$ whenever $y$ is distinguishable from $x$. It is transitive because if $w \equiv_L x$ and $x \equiv_L y$, then for each $z$, $wz \in L$ iff $xz \in L$ and $xz \in L$ iff $yz \in L$, hence $wz \in L$ iff $yz \in L$, and so $w \equiv_L y$.

**1.35** a. We prove this assertion by contradiction. Let $M$ be a $k$-state DFA that recognizes $L$. Suppose for a contradiction that $L$ has index greater than $k$. That means some set $X$ with more than $k$ elements is pairwise distinguishable by $L$. Because $M$ has $k$ states, the pigeonhole principle implies that $X$ contains two distinct strings $x$ and $y$ where $\delta(q_0, x) = \delta(q_0, y)$. Here $\delta(q_0, x)$ is the state that $M$ is in after starting in the start state $q_0$ and reading

input string $x$. Then, for any string $z \in \Sigma^*$, $\delta(q_0, xz) = \delta(q_0, yz)$. Therefore either both $xz$ and $yz$ are in $L$ or neither are in $L$. But then $x$ and $y$ aren't distinguishable by $L$, contradicting our assumption that $X$ is pairwise distinguishable by $L$.

b. Let $X = \{s_1, \ldots, s_k\}$ be pairwise distinguishable by $L$. We construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ with $k$ states recognizing $L$. Let $Q = \{q_1, \ldots, q_k\}$, and define $\delta(q_i, a)$ to be $q_j$ where $s_j \equiv_L s_i a$ (the relation $\equiv_L$ is defined in problem 1.34). Note that $s_j \equiv_L s_i a$ for some $s_j \in X$ otherwise $X \cup s_i a$ would have $k+1$ elements and would be pairwise distinguishable by $L$ and that contradicts our assumption that $L$ has index $k$. Let $F = \{q_i | \ s_i \in L\}$. Let the start state, $q_0$, be the $q_i$ such that $s_i \equiv_L \varepsilon$. $M$ is constructed so that, for any state $q_i$, $\{s | \ \delta(q_0, s) = q_i\} = \{s | \ s \equiv_L s_i\}$. Hence $M$ recognizes $L$.

c. Suppose $L$ is regular and let $k$ be the number of states in a DFA recognizing $L$. Then from part (a) $L$ has index at most $k$. Conversely, if $L$ has index $k$, then by part (b) it is recognized by a DFA with $k$ states, and thus is regular. To show that the index of $L$ is the size of the smallest DFA accepting it, suppose that $L$'s index is *exactly* $k$. Then, by part (b), there is a $k$-state DFA accepting $L$. That is the smallest such DFA since if it were any smaller, then we could show by part (a) that the index of $L$ is less than $k$.

**1.36** Assume to the contrary that $ADD$ is regular. Let $p$ be the pumping length given by the pumping lemma. Choose $s$ to be the string $1^p{=}0^p{+}1^p$, which is a member of $ADD$. Because $s$ has length greater than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, satisfying the conditions of the lemma. By the third condition in the pumping lemma have that $|xy| \leq p$, it follows that $y$ is $1^k$ for some $k \geq 1$. Then $xy^2z$ is the string $1^{p+k}{=}0^p{+}1^p$, which is not a member of $ADD$, violating the pumping lemma. Hence $ADD$ isn't regular.

**1.37** **Note:** The version of the problem that appears in the first printing is incorrect. The correct version (which appears in the second and subsequent printings) is:

Show that the language $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | \ i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}$ satisfies the three conditions of the pumping lemma even though it is not regular. Explain why this fact does not contradict the pumping lemma.

Language $F$ satisfies the conditions of the pumping lemma using pumping length 2. If $s \in F$ is of length 2 or more we show that it can be pumped by considering four cases, depending on the number of a's that $s$ contains.

i) If $s$ is of the form $\mathtt{b}^*\mathtt{c}^*$, let $x = \varepsilon$, $y$ be the first symbol of $s$, and let $z$ be the rest of $s$.

ii) If $s$ is of the form $\mathtt{ab}^*\mathtt{c}^*$, let $x = \varepsilon$, $y$ be the first symbol of $s$, and let $z$ be the rest of $s$.

iii) If $s$ is of the form $\mathtt{aab}^*\mathtt{c}^*$, let $x = \varepsilon$, $y$ be the first two symbols of $s$, and let $z$ be the rest of $s$.

iv) If $s$ is of the form $\mathtt{aaa^*b^*c^*}$, let $x = \varepsilon$, $y$ be the first symbol of $s$, and let $z$ be the rest of $s$.

In each case, the strings $xy^iz$ are members of $F$ for every $i \geq 0$. Hence $F$ satisfies the conditions of the pumping lemma. However, $F$ is clearly not regular, because the nonregular language $\{\mathtt{ab^nc^n} \mid n \geq 0\}$ is the same as $F \cap \mathtt{ab^*c^*}$, and the regular languages are closed under intersection. The pumping lemma is not violated because it states only that regular languages satisfy the three conditions, and it doesn't state that nonregular languages fail to satisfy the three conditions.

**1.38** The objective of this problem is for the student to pay close attention to the exact formulation of the pumping lemma.

**a.** The minimum pumping length is 4. The string $s = 000 \in 0001^*$ cannot be pumped, and any string in $0001^*$ of length 4 or more contains a 1 and hence can be pumped by dividing it so that $x = 0$, $y = 1$, and $z$ is the rest.

**b.** The minimum pumping length is 1. The string $\varepsilon$ cannot be pumped because of condition 2 in the pumping lemma and hence 0 is not a pumping length. Any string in $0^*1^*$ of length 1 or more contains a 0 or a 1 and hence can be pumped by dividing it so that $x = \varepsilon$, $y = 0$ or 1, and $z$ is the rest, so 1 is a pumping length.

**c.** The minimum pumping length is 1. The pumping length cannot be 0, as in part (b). Any string in $(01)^*$ of length 1 or more contains 01 and hence can be pumped by dividing it so that $x = \varepsilon$, $y = 01$, and $z$ is the rest.

**d.** The minimum pumping length is 3. The string 01 cannot be pumped, so 2 is not a pumping length. All strings in the language of length 3 or more can be pumped, (note that there aren't any such string, so the statement is true vacuously) so 3 is a pumping length.

**e.** The minimum pumping length is 1. String $\varepsilon$ cannot be pumped because of condition 2 in the pumping lemma and hence 0 is not a pumping length. The language has no strings of length 1 or more so 1 is a pumping length. (the conditions hold vacuously).

**1.39** Let $A_k = \Sigma^*0^{k-1}0^*$. A DFA with $k$ states can recognize $A_k$. In addition to the start state, it has one state for each number of 0s it has read since the last 1. After $k-1$ 0s the machine enters a accept state whose transitions are self-loops.

In any DFA with fewer than $k$ states, two of the $k$ strings 1, 10, ... , $10^{k-1}$ must cause the machine to enter the same state, by the pigeon hole principle. But then, if we add to both of these strings enough 0s to cause the longer of these two strings to have exactly $k-1$ 0s, the two new strings will still cause the machine to enter the same state, but one of these strings is in $A_k$ and the other is not. Hence, the machine must fail to produce the correct accept/reject response on one of these strings.

**1.40** Let $A = \{2^n \mid n \geq 0\}$. The set $B_2(A)$ consists of all strings of the form $10^*$ which is obviously regular. We show by contradiction that $B = B_3(A)$ isn't regular. Suppose that it is. Then there exists a pumping length $p$ such that for any string $s \in B$ with $|s| \geq p$, we can break $s$ up into strings $x$, $y$, and $z$ satisfying the conditions of the lemma. Consider the string $s \in B$ that is the base 3 representation of $2^l$ where we choose $l$ large enough to force $s$ to have length at least $p$. Then $s = xyz$ satisfying the conditions of the lemma. In particular, $xyz \in B$, $xy^2z \in B$, and $xz \in B$. Now, define the length of $y$ to be $m$ and define the length of $z$ to be $n$. Recall that $m$ is strictly greater than 0. Treat strings as base 3 representations of numbers in the following equations:

i) $xz = x3^n + z$. (Call this quantity $q_0$)
ii) $xyz = x3^{m+n} + y3^n + z$. (Call this quantity $q_1$)
iii) $xyyz = x3^{2m+n} + y3^{m+n} + y3^n + z$. (Call this quantity $q_2$)

We now derive a contradiction algebraically. Using equations (i), (ii), and (iii), we have

$$\frac{q_2 - q_1}{q_1 - q_0} = \frac{x3^{2m+n} + (y-x)3^{m+n}}{x3^{m+n} + (y-x)3^n} = 3^m.$$

We know from the pumping lemma that $xz, xyz, xyyz \in B$ so it follows that $q_0 = 2^{l_0}$ and $q_2 = 2^{l_2}$, for some $l_0, l_2 \geq 0$. Because $s = xyz$ and $s$ is the base 3 representation of $2^l$, we have $q_1 = 2^l$. Moreover, $l_0 < l < l_2$. Thus,

$$\frac{q_2 - q_1}{q_1 - q_0} = \frac{2^{l_2} - 2^l}{2^l - 2^{l_0}} = \frac{2^{l_2 - l_0} - 2^{l - l_0}}{2^{l - l_0} - 1} = \frac{2^{l - l_0}(2^{l_2 - l} - 1)}{2^{l - l_0} - 1}.$$

Setting these two expression equal we obtain

$$\frac{2^{l - l_0}(2^{l_2 - l} - 1)}{2^{l - l_0} - 1} = 3^m \quad \text{and thus} \quad 2^{l - l_0}(2^{l_2 - l} - 1) = 3^m(2^{l - l_0} - 1).$$

The left-hand side of this equation is an even number and the right-hand side of this equation is an odd number. Hence we arrive at a contradiction.

**1.41** The language $D$ can be written in another way, namely, $0\Sigma^*0 \cup 1\Sigma^*1$, which is obviously regular.

**1.42** Given an NFA $M$ recognizing $A$ we construct an NFA $N$ accepting $A_{\frac{1}{2}-}$ using the following idea. $M$ keeps track two states in $N$ using two "fingers". As it reads each input symbol $N$ uses one finger to simulate $M$ on that symbol. At the same time it from an accept state on a guessed symbol. Simultaneously, $M$ uses the other finger to run $M$ backwards from an accept state on a guessed symbol. $N$ accepts whenever the forward simulation and the backward simulation are in the same state, that is, whenever the two fingers re together. At those points we are sure that $N$ has found a string

where another string of the same length can be appended to yield a member of $A$, precisely the definition of $A_{\frac{1}{2}-}$.

Formally, we assume for simplicity that the NFA $M$ recognizing $A$ has a single accept state as guaranteed by Exercise 1.9. Let $M = (Q, \Sigma, \delta, q_0, q_{\text{accept}})$. Construct NFA $N = (Q', \Sigma, \delta', q_0', F')$ recognizing the first halves of the strings in $A$ as follows:

i) $Q' = Q \times Q$.
ii) For $q, r \in Q$ define $\delta'((q,r),a) = \{(u,v) |\ u \in \delta(q,a)$ and $r \in \delta(v,b)$ for some $b \in \Sigma\}$.
iii) $q_0' = (q_0, q_{\text{accept}})$.
iv) $F' = \{(q,q) |\ q \in Q\}$.

**1.43** Let $A = \{0^*\#1^*\}$. Thus, $A_{\frac{1}{3}-\frac{1}{3}} \cap \{0^*1^*\} = \{0^n1^n |\ n \geq 0\}$. Regular sets are closed under intersection, and $\{0^n1^n |\ n \geq 0\}$ is not regular, so $A_{\frac{1}{3}-\frac{1}{3}}$ is not regular.

**1.44** Let $E_n = \{x \in \{0,1\}^* |$ the $(n-1)$st symbol from the right in $x$ is $1\}$ for $n \geq 2$. Constructing an $n$-state NFA for $E_n$ is easy.

Let $M$ be a DFA recognizing $E_n$. We claim that $M$ has at least $2^{n-1}$ states. Let $S$ be the set of all strings of length $n-1$. For each $y$ in $S$ let $q_y$ be the state that $M$ is in after starting at the start state and the reading $y$. If $y \neq z$, then $q_y \neq q_z$, otherwise $M$ fails to recognize the $E_n$ for the following reason. Say that $y$ and $z$ differ on the $i$th bit. Then one of the strings $y0^{i-1}$ and $z0^{i-1}$ is in $E_n$ and the other one isn't. But $M$ has the same behavior on both, so $M$ fails to recognize $E_n$. Hence, $M$ has at least as many states a $S$ has strings, and so $M$ has at least $2^{n-1}$ states.

# Chapter 2

**2.1** Here are the derivations (but not the parse trees).

a. $E \Rightarrow T \Rightarrow F \Rightarrow \mathtt{a}$
b. $E \Rightarrow E{+}T \Rightarrow T{+}T \Rightarrow F{+}T \Rightarrow F{+}F \Rightarrow \mathtt{a}{+}F \Rightarrow \mathtt{a}{+}\mathtt{a}$
c. $E \Rightarrow E{+}T \Rightarrow E{+}T{+}T \Rightarrow T{+}T{+}T \Rightarrow F{+}T{+}T \Rightarrow F{+}F{+}T \Rightarrow F{+}F{+}F \Rightarrow \mathtt{a}{+}F{+}F \Rightarrow \mathtt{a}{+}\mathtt{a}{+}F \Rightarrow \mathtt{a}{+}\mathtt{a}{+}\mathtt{a}$
d. $E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (F) \Rightarrow ((E)) \Rightarrow ((T)) \Rightarrow ((F)) \Rightarrow ((\mathtt{a}))$

**2.2** a. First, we show that both $A$ and $B$ are context-free. It's not hard to see that the following grammar recognizes $A$:

$$
\begin{aligned}
S &\rightarrow RT \\
R &\rightarrow \mathtt{a}R \mid \varepsilon \\
T &\rightarrow \mathtt{b}T\mathtt{c} \mid \varepsilon
\end{aligned}
$$

The following grammar recognizes $B$:

$$\begin{aligned} S &\to TR \\ T &\to \mathtt{a}T\mathtt{b} \mid \varepsilon \\ R &\to \mathtt{c}R \mid \varepsilon \end{aligned}$$

So, both $A$ and $B$ are context-free languages. Now, observe that $A \cap B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. We know from Example 2.20 that this language is not context-free. We have found two context free languages whose intersection is not context-free. Therefore context-free languages are not closed under intersection.

**b.** First, the context-free languages are closed under the union operation. Let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ be two arbitrary context free grammars. We construct a grammar $G$ that recognizes their union. Formally, $G = (V, \Sigma, R, S)$ where:

i) $V = V_1 \cup V_2$

ii) $R = R_1 \cup R_2 \cup \{S \to S_1,\ S \to S_2\}$
(Here we assume that $R_1$ and $R_2$ are disjoint, otherwise we change the variable names to ensure disjointness)

Next, we show that the CFGs are not closed under complementation. Assume, for a contradiction, that the CFGs are closed under complementation. Then, if $G_1$ and $G_2$ are context free grammars, it would follow that $\overline{L(G_1)}$ and $\overline{L(G_2)}$ are context free. We previously showed that context-free languages are closed under union and so $\overline{L(G_1)} \cup \overline{L(G_1)}$ is context free. That, by our assumption, implies that $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}}$ is context free. But by DeMorgan's laws, $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}} = L(G_1) \cap L(G_2)$. However, if $G_1$ and $G_2$ are chosen as in part (a), $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}}$ isn't context free. This contradiction shows that the context-free languages are not closed under complementation.

**2.3**

**a.** The variables of $G$ are: $R, X, S, T$. The terminals of $G$ are: $\mathtt{a}, \mathtt{b}$. The start variable is $R$.

**b.** Three strings in $G$ are: $\mathtt{ab}, \mathtt{ba}$, and $\mathtt{aab}$.

**c.** Three strings not in $G$ are: $\mathtt{a}, \mathtt{b}$, and $\varepsilon$.

**d.** False.

**e.** True.

**f.** False.

**g.** True.

**h.** True.

**i.** False.

**j.** True.

**k.** True.

**l.** False.

**m.** $L(G)$ consists of all strings over $\mathtt{a}$ and $\mathtt{b}$ that are not palindromes.

**2.4** **a.** $S \to R1R1R1R$
$R \to 0R \mid 1R \mid \varepsilon$

**b.** $S \to 0R0 \mid 1R1 \mid \varepsilon$
$R \to 0R \mid 1R \mid \varepsilon$

**c.** $S \to 0 \mid 1 \mid 00S \mid 01S \mid 10S \mid 11S$

**d.** $S \to 0 \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$

**e.** $S \to B1B$
$B \to BB \mid 0B1 \mid 1B0 \mid 1 \mid \varepsilon$
This CFG works because $B$ generates all strings that have at least as many 1s as 0s, and so $S$ forces an extra 1.

**f.** $S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$

**g.** $S \to S$

**2.5** **a.** This set is regular, so the PDA doesn't even need to use its stack. The PDA scans the string and uses its finite control to maintain a counter which counts up to 3. It keeps track of the number of 1's it has seen using this counter. The PDA accepts the moment it sees three ones.

**b.** This set is also regular. The PDA scans the string and keep track of the first and last symbol in its finite control. If they are the same, it accepts, otherwise it rejects.

**c.** Again, this set is regular. The PDA scans the string and keep track of the length (modulo 2) using its finite control. If the length is 1 (modulo 2) it accepts, otherwise it rejects.

**d.** The PDA scans across the string and push the symbols onto the stack. At some point it nondeterministically guesses where the middle is. It looks at the middle symbol. If that symbol is a 1, it rejects. Otherwise, it scans the rest of the string, and for each character scanned, it pops one element off of its stack. If the stack is empty before it reaches the end of the input, it rejects. If the stack is empty when it finishes reading the input (i.e. it correctly guessed the middle) then it accepts.

**e.** The PDA scans across the input. If it sees a 1 and its top stack symbol is a 0, it pops the stack. Similarly, if it scans a 0 and its top stack symbol is a 1, it pops the stack. In all other cases, it pushes the input symbol onto the stack. After it scans the input, if there is a 1 on top of the stack, it accepts. Otherwise it rejects.

**f.** The PDA begins by scanning across the string and pushing each symbol onto its stack. At some point it nondeterministically guesses when it has reached the middle. It also nondeterministically guesses if string has odd length or even length. If it guessed even, then it pushes the current symbol it's reading onto the stack (recall that the PDA has guessed that this symbol is the middle symbol). If it guesses that string has odd length, it goes to the next input symbol without changing the stack. Now, it scans the rest of the string, and it compares each symbol it scans to the symbol on the top of the stack. If they are the same, it pops the stack, and continues scanning. If they are different, it rejects. If the stack is empty before it finishes reading all the

input, it rejects. If the stack becomes empty just after it reaches the end of the input then it accepts. In all other cases, it rejects.

g. The PDA simply rejects immediately.

**2.6** a. $S \to S\mathtt{a}S\mathtt{a}S\mathtt{b}S \mid S\mathtt{a}S\mathtt{b}S\mathtt{a}S \mid S\mathtt{b}S\mathtt{a}S\mathtt{a}S \mid \varepsilon$

b. $S \to X\mathtt{b}X\mathtt{a}B \mid T \mid U$
$T \to \mathtt{a}T\mathtt{b} \mid T\mathtt{b} \mid \mathtt{b}$
$U \to \mathtt{a}U\mathtt{b} \mid \mathtt{a}U \mid \mathtt{a}$
$X \to \mathtt{a} \mid \mathtt{b}$

c. $S \to TX$
$T \to 0T0 \mid 1T1 \mid \#X$
$X \to 0X \mid 1X \mid \varepsilon$

d. $S \to UPV$
$P \to \mathtt{a}P\mathtt{a} \mid \mathtt{b}P\mathtt{b} \mid T \mid \varepsilon$
$T \to \#MT \mid \#$
$U \to M\#U \mid \varepsilon$
$V \to \#MV \mid \varepsilon$
$M \to \mathtt{a}M \mid \mathtt{b}M \mid \varepsilon$
Note that we need to allow for the case when $i = j$, that is, some $x_i$ is a palindrome. Also, $\varepsilon$ is in the language since it's a palindrome.

**2.7** a. The PDA uses its stack to keep a count of the number of a's minus twice the number of b's. It enters an accepting state whenever this count is 0. In more detail, it operates as follows. It reads the next input symbol. If it is an a and the top of the stack is a b, it pops the b. If it is an a and the top of is either an a or empty, it pushes the a. If the next input symbol is a b, then repeat the following sentence twice. If the top if the stack is an a, pop the stack, otherwise push the b.

b. The PDA scans across the string. It pushes the a's it reads onto the stack until it encounters a b. Then, for each b it reads, it pops one a off the stack. If the machine encounters an a after a b, it accepts (by entering an accept state and moving its head to the end of the input). If the stack becomes empty before the end of the input is reached, of if the end of the input is reached without the stack becoming empty, the machine accepts. Otherwise it rejects.

c. The PDA scans across the input string and pushes every symbol it reads until it reads a #. If # is never encountered, reject. Then, it skips over part of the input, nondeterministically deciding when to stop skipping. Then, it compares the input symbols it next read with the symbols it pops off the stack. At any disagreement, or if the input finishes while the stack is nonempty, this branch of the computation rejects. If the stack becomes empty, the machine accepts.

d. The PDA uses its nondeterminism to guess the values of $i$ and $j$ and uses its stack to compare $x_i$ with $x_j^{\mathcal{R}}$. In more detail, first it nondeterministically branches to the procedures in the next two paragraphs.

The machine begins by nondeterministically skipping over a part (possibly empty) of the input until it arrives at a symbol which is either at the beginning of the input string or is immediately to the right of a #. Then, the machine continues to read input symbols, while pushing them onto the stack. During this reading and popping phase, two cases arise. In one case, if an input # is read, the machine nondeterministically skips to another # in the input and continues reading inputs symbols, now comparing them with symbols popped off the stack. If all agree and the stack becomes empty at the same time as either the end of the input of # symbol is reached, accept. However, if a disagreement occurs, or the stack becomes empty before either the end of the input of # symbol is reached, or either the end of the input of # symbol is reached without the stack becoming empty, reject on this branch of the nondeterminism. In the other case, during the reading and popping phase, the machine may nondeterministically chose to enter the reading and popping phase, as just described. Then it continues to operate as in that case.

**2.8** Here is one derivation:

⟨SENTENCE⟩ ⇒
⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩ ⇒
⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩ ⇒
⟨CMPLX-NOUN⟩⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩ ⇒
⟨ARTICLE⟩⟨NOUN⟩⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩ ⇒
The boy ⟨VERB⟩⟨NOUN-PHRASE⟩⟨PREP-PHRASE⟩ ⇒
The boy ⟨VERB⟩⟨NOUN-PHRASE⟩⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches ⟨NOUN-PHRASE⟩⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches ⟨CMPLX-NOUN⟩⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches ⟨ARTICLE⟩⟨NOUN⟩⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches the girl with ⟨CMPLX-NOUN⟩ ⇒
The boy touches the girl with ⟨ARTICLE⟩⟨NOUN⟩ ⇒
The boy touches the girl with the flower

Here is another derivation:

⟨SENTENCE⟩ ⇒
⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩ ⇒
⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩ ⇒
⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩ ⇒
The boy ⟨VERB-PHRASE⟩ ⇒
The boy ⟨CMPLX-VERB⟩ ⇒
The boy ⟨VERB⟩⟨NOUN-PHRASE⟩ ⇒
The boy touches ⟨NOUN-PHRASE⟩ ⇒
The boy touches ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩ ⇒
The boy touches ⟨ART⟩⟨NOUN⟩⟨PREP-PHRASE⟩ ⇒
The boy touches the girl ⟨PREP-PHRASE⟩ ⇒
The boy touches the girl ⟨PREP⟩⟨CMPLX-NOUN⟩ ⇒
The boy touches the girl with ⟨CMPLX-NOUN⟩ ⇒

The boy touches the girl with ⟨ARTICLE⟩⟨NOUN⟩ ⇒
The boy touches the girl with the flower

Each of these derivations corresponds to a different English meaning. In the first derivation, the sentence means that the boy used the flower in order to touch the girl. In the second derivation, the girl happens to be holding the flower when the boy touches her.

**2.9** A CFG $G$ that generates $A$ is given as follows:

$G = (V, \Sigma, R, S)$.

$V$ is $\{S, E_{ab}, E_{bc}, C, A\}$ and $\Sigma$ is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. The rules are

$$\begin{aligned} S &\rightarrow E_{ab}C \mid AE_{bc} \\ E_{ab} &\rightarrow \mathtt{a}E_{ab}\mathtt{b} \mid \varepsilon \\ E_{bc} &\rightarrow \mathtt{b}E_{bc}\mathtt{c} \mid \varepsilon \\ C &\rightarrow C\mathtt{c} \mid \varepsilon \\ A &\rightarrow A\mathtt{a} \mid \varepsilon \end{aligned}$$

Initially substituting $E_{ab}C$ for $S$ generates any string with an equal number of as and bs followed by any number of cs. Initially substituting $E_{bc}$ for $S$ generates any string with an equal number of bs and cs prepended by any number of as.

The grammar is ambiguous. Consider the string $\varepsilon$. On the one hand, it can be derived by choosing $E_{ab}$C with each of $E_{ab}$ and $C$ yielding $\varepsilon$. On the other hand, $\varepsilon$ can be derived by choosing $AE_{bc}$ with each of $A$ and $E_{bc}$ yielding $\varepsilon$. In general, any string $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$ with $i = j = k$ can be derived ambiguously in this grammar.

**2.10** Informal description of a PDA that recognizes $A$ in Exercise 2.9:

1. Nondeterministically branch to either step 2 or step 6.
2. Read and push a's.
3. Read b's, while popping a's.
4. If b's finish when stack is empty, skip c's on input and *accept*.
5. Skip a's on input.
6. Read and push b's.
7. Read c's, while popping b's.
8. If c's finish when stack is empty, *accept*.

**2.11** Informal description of a PDA that recognizes the CFG in Exercise 2.1:

1. Place the marker symbol \$ and the start variable $E$ on the stack.
2. Repeat the following steps forever.

3. If the top of stack is the variable $E$, pop it and nondeterministically push either $E$+$T$ or $T$ into the stack.
4. If the top of stack is the variable $T$, pop it and nondeterministically push either $T$×$F$ or $F$ into the stack.
5. If the top of stack is the variable $F$, pop it and nondeterministically push either ($E$) or a into the stack.
6. If the top of stack is a terminal symbol, read the next symbol from the input and compare it to the terminal in the stack. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
7. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

The formal definition of the equivalent PDA is $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where Here $Q = \{q_1, q_2\}$; $\Sigma = \{+, \times, (, ), \mathtt{a}\}$; and $\Gamma = \{E, T, F\} \cup \Sigma$; $F = \{q_2\}$. The transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is given as follows.

$$\delta(q,x,y) = \begin{cases} \{(q_2, \varepsilon)\} & \text{if } q = q_1, x = \varepsilon, y = \$ \\ \{(q_1, E{+}T), (q_1, T)\} & \text{if } q = q_1, x = \varepsilon, y = E \\ \{(q_1, T{\times}F), (q_1, F)\} & \text{if } q = q_1, x = \varepsilon, y = T \\ \{(q_1, (E)), (q_1, \mathtt{a})\} & \text{if } q = q_1, x = \varepsilon, y = F \\ \{(q_1, \varepsilon)\} & \text{if } q = q_1, x = y \end{cases}$$

**2.12** Informal description of a PDA that recognizes the CFG in Exercise 2.3:

1. Place the marker symbol \$ and the start variable $R$ on the stack.
2. Repeat the following steps forever.
3. If the top of stack is the variable $R$, pop it and nondeterministically push either $XRX$ or $S$ into the stack.
4. If the top of stack is the variable $S$, pop it and nondeterministically push either a$T$b or b$T$a into the stack.
5. If the top of stack is the variable $T$, pop it and nondeterministically push either $XTX$, $X$ or $\varepsilon$ into the stack.
6. If the top of stack is the variable $X$, pop it and nondeterministically push either a or b into the stack.
7. If the top of stack is a terminal symbol, read the next symbol from the input and compare it to the terminal symbol in the stack. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
8. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

The formal definition of the equivalent PDA is $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where $Q = \{q_1, q_2\}$; $\Sigma = \{\mathtt{a}, \mathtt{b}\}$; $\Gamma = \{R, S, T, X\} \cup \Sigma$; and $F = \{q_2\}$. The transition

function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is given as follows.

$$\delta(q,x,y) = \begin{cases} \{(q_2,\varepsilon)\} & \text{if } q = q_1, x = \varepsilon, y = \$ \\ \{(q_1, XRX), (q_1, S)\} & \text{if } q = q_1, x = \varepsilon, y = R \\ \{(q_1, \mathtt{a}T\mathtt{b}), (q_1, \mathtt{b}T\mathtt{a})\} & \text{if } q = q_1, x = \varepsilon, y = S \\ \{(q_1, XTX), (q_1, X), (q_1, \varepsilon)\} & \text{if } q = q_1, x = \varepsilon, y = T \\ \{(q_1, \mathtt{a}), (q_1, \mathtt{b})\} & \text{if } q = q_1, x = \varepsilon, y = X \\ \{(q_1, \varepsilon)\} & \text{if } q = q_1, x = y \end{cases}$$

**2.13** a. $L(G)$ is the set of strings of 0s and #s that either contain exactly 2 #s and any number of 0s, or contain exactly 1 # and the number of 0s to the right of the # is twice the number of 0s to the left.

b. Assume $L(G)$ is regular. Let $A = L(G) \cap 0^*\#0^*$. If $L(G)$ is regular, so is $A$. Let $p$ be the pumping length for $A$ given by the pumping lemma for regular languages. Consider the string $w = 0^p\#0^{2p}$. Because $|w| > p$ and $w \in A$, the pumping lemma that $w = xyz$ such that $|xy| \leq p$, $y \neq \varepsilon$ and $xy^iz \in L(G)\ \forall i \geq 0$. We investigate all possible ways of cutting $w$ and prove that such a cut cannot exist.

i) $x$ contains the character #. In this case, $y$ is to the right of #. Pumping it down makes the number of 0s on the right less than twice of the number of 0s on the left.

ii) $y$ contains the character #. Pumping $y$ down makes the resulting string contain no #s.

iii) $z$ contains the character #. In this case, $y$ is to the left of #. Pumping $y$ down makes the number of 0s on the right more than twice of the number of 0s on the left.

In each case the resulting string doesn't belong to $A$. Hence $A$ fails the pumping lemma, so it cannot be regular, and neither can $L(G)$.

**2.14** The equivalent CFG in Chomsky normal form is given as follows.

$$\begin{aligned} S_0 &\to AB \mid CC \mid BA \mid BD \mid BB \mid \varepsilon \\ A &\to AB \mid CC \mid BA \mid BD \mid BB \\ B &\to CC \\ C &\to 0 \\ D &\to AB \end{aligned}$$

**2.15** Let $L(G_1)$ and $L(G_2)$ be CFGs where $G_1$ and $G_2$ are defined as follows. (Here we assume that that the sets of variables are disjoint.) let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$.

We construct a new CFG $G_\cup$ where $L(G_\cup) = L(G_1) \cup L(G_2)$. Let $S'$ be a new variable that is neither in $V_1$ nor in $V_2$.

Let $G_\cup = (V_1 \cup V_2 \cup \{S'\},\ \Sigma,\ R_1 \cup R_2 \cup \{r_0\},\ S')$, where $r_0$ is $S' \to S_1 \mid S_2$.

We construct CFG $G_\circ$ that generates $L(G_1) \circ L(G_2)$ as in $G_\cup$ by changing $r_0$ in $G_\cup$ into $S' \to S_1 S_2$.

To construct CFG $G_*$ that generates the language $L(G_1)^*$, let $S'$ be a new variable not in $V_1$ and make it the starting variable. Let $r_0$ be $S' \to S'S_1 \mid \varepsilon$ be a new rule in $G_*$.

**2.16** Let $A$ be a regular language generated by regular expression $R$. If $R$ is one of the atomic regular expressions $b$, for $b \in \Sigma$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to b\}, S)$. If $R$ is the atomic regular expression $\varepsilon$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to \varepsilon\}, S)$. If $R$ is the atomic regular expressions $\emptyset$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to S\}, S)$.

If $R$ is a regular expression composed of smaller regular expressions combined with a regular operation, use the result of Problem 2.15 to construct an equivalent CFG out of the CFLs that are equivalent to the smaller expressions.

**2.17** **a.** Let $C$ be a context-free language and $R$ be a regular language. Let $P$ be the PDA that recognizes $C$, and $D$ be the DFA that recognizes $R$. If $Q$ is the set of states of $P$ and $Q'$ is the set of states of $D$, we construct a PDA $P'$ that recognizes $C \cap R$ with the set of states $P \times Q$. $P'$ will do what $P$ does and also keep track of the states of $D$. It accepts a string $w$ if and only if it stops at a state $q \in F_P \times F_D$, where $F_P$ is the set of accept states of $P$ and $F_D$ is the set of accept states of $D$. Since $C \cap R$ is recognized by $P'$, it is context free.

**b.** Let $R$ be the regular language $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$. If $A$ were $A$ a CFL then $A \cap R$ would be a CFL by part (a). However, $A \cap R = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ and Example 2.20 proves $A \cap R$ is not context-free. Thus $A$ is not a CFL.

**2.18** **a.** Let $A = \{0^n1^n0^n1^n \mid n \geq 0\}$. Let $p$ be the pumping length given by the pumping lemma. We show that $s = 0^p1^p0^p1^p$ cannot be pumped. Let $s = uvxyz$.

If either $v$ or $y$ contain more than one type of alphabet symbol, $uv^2xy^2z$ does not contain the symbols in the correct order. Hence it cannot be a member of $A$. If both $v$ and $y$ contain (at most) one type of alphabet symbol, $uv^2xy^2z$ contains runs of 0's and 1's of unequal length. Hence it cannot be a member of $A$. Because $s$ cannot be pumped without violating the pumping lemma conditions, $A$ is not context free.

**b.** Let $B = \{0^n\#0^{2n}\#0^{3n} \mid n \geq 0\}$. Let $p$ be the pumping length given by the pumping lemma. Let $s = 0^p\#0^{2p}\#0^{3p}$. We show that $s = uvxyz$ cannot be pumped.

Neither $v$ nor $y$ can contain #, otherwise $xv^2wy^2z$ contains more than two #s. Therefore, if we divide $s$ into three segments by #'s: $0^p, 0^{2p}$, and $0^{3p}$, at least one of the segments is not contained within either $v$ or $y$. Hence, $xv^2wy^2z$ is not in $B$ because the $1:2:3$ length ratio of the segments is not maintained.

c. Let $C = \{w\#x \mid w \text{ is a substring of } x, \text{ where } w, x \in \{\mathtt{a},\mathtt{b}\}^*\}$. Let $p$ be the pumping length given by the pumping lemma. Let $s = \mathtt{a}^p\mathtt{b}^p\#\mathtt{a}^p\mathtt{b}^p$. We show that the string $s = uvxyz$ cannot be pumped.
Neither $v$ nor $y$ can contain #, otherwise $uv^0xy^0z$ does not contain # and therefore is not in $C$. If both $v$ and $y$ are nonempty and occur on the left-hand side of the #, the string $uv^2xy^2z$ cannot be in $C$ because it is longer on the left-hand side of the #. Similarly, if both strings occur on the right-hand side of the #, the string $uv^0xy^0z$ cannot be in $C$, because it is again longer on the left-hand side of the #. If only one of $v$ and $y$ is nonempty (both cannot be nonempty), treat them as if both occurred on the same side of the # as above.
The only remaining case is where both $v$ and $y$ are nonempty and straddle the #. But then $v$ consists of b's and $y$ consists of a's because of the third pumping lemma condition $|vxy| \leq p$. Hence, $uv^2xy^2z$ contains more b's on the left-hand side of the #, so it cannot be a member of $C$.

d. Let $D = \{x_1\#x_2\#\cdots\#x_k \mid k \geq 2, \text{ each } x_i \in \{\mathtt{a},\mathtt{b}\}^*, \text{and for some } i{\neq}j, x_i{=}x_j\}$. Let $p$ be the pumping length given by the pumping lemma. Let $s = \mathtt{a}^p\mathtt{b}^p\#\mathtt{a}^p\mathtt{b}^p$. We show that $s = uvxyz$ cannot be pumped. Use the same reasoning as in part (c).

**2.19** **Note:** The version of the problem that appears in the first printing is incorrect. The correct version (which appears in the second and subsequent printings) is: Show that the language $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}$ satisfies the three conditions of the pumping lemma even though it is not regular. Explain why this fact does not contradict the pumping lemma.
Every rule of a Chomsky normal form grammar is of the form

$$A \to BC$$
$$A \to \mathtt{a}$$

(except possibly for the rule $S \to \varepsilon$, where $S$ is the start variable. This rule isn't used in the derivation of $w$ because $n \geq 1$.)
Consider a derivation of $w$. Each application of a rule of the form $A \to BC$ increases the length of the string by 1. So we have $n - 1$ steps here. Besides that, we need exactly $n$ applications of terminal rules ($A \to a$) to convert the variables into terminals. Therefore, exactly $2n - 1$ steps are required.

**2.20** **Note:** The version of the problem that appears in the first printing is incorrect. The correct version (which appears in the second and subsequent printings) replaces the phrase "more than $b$ steps" with the phrase "at least $2^b$ steps".
Let $G$ be a CFG in Chomsky normal form that contains $b$ variables. Assume G generates a string $w$ using a derivation with at least $2^b$ steps. Let $n$ be the length of $w$. By the results of Problem 2.20, $n \geq \frac{2^b+1}{2} > 2^{b-1}$.

Consider a parse tree of $w$. The right-hand side of each rule contains at most two variables, so each node of the parse tree has at most two children. Additionally, the length of $w$ is at least $2^b$, so the parse tree of $w$ must have height at least $b+1$ to generate a string of length at least $2^b$. Hence, the tree contains a path with at least $b+1$ variables, and therefore some variable is repeated on that path. Using a surgery on trees argument identical to the one used in the proof of the CFL pumping lemma, we can now divide w into pieces $uvxyz$ where $uv^ixy^iz \in G$ for all $i \geq 0$. Therefore, $L(G)$ is infinite.

**2.21** **a.** To see that $G$ is ambiguous, note that the string

`if condition then if condition then a:=1 else a:=1`

has two different leftmost derivations (and hence parse trees):

⟨STMT⟩
→ ⟨IF-THEN⟩
→ `if condition then` ⟨STMT⟩
→ `if condition then` ⟨IF-THEN-ELSE⟩
→ `if condition then if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else a:=1`

and

⟨STMT⟩
→ ⟨IF-THEN-ELSE⟩
→ `if conditon then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if conditon then` ⟨IF-THEN⟩ `else` ⟨STMT⟩
→ `if conditon then if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else a:=1`

**b.** The ambiguity in part a) arises because the grammar allows matching an `else` both to the nearest and to the farthest `then`. To avoid this ambiguity we construct a new grammar that only permits the nearest match, by disabling derivations which introduce an ⟨IF-THEN⟩ before an `else`. This grammar has two new variables: ⟨E-STMT⟩ and ⟨E-IF-THEN-ELSE⟩, which work just like their non-⟨E⟩ counterparts except that they cannot generate the dangling ⟨IF-THEN⟩. The rules of the new grammar are the same as for the old grammar except we remove the ⟨IF-THEN-ELSE⟩ rules and add the following new rules:

⟨E-STMT⟩ → ⟨ASSIGNMENT⟩ | ⟨BEGIN-END⟩ | ⟨E-IF-THEN-ELSE⟩
⟨E-IF-THEN-ELSE⟩ → `if conditon then` ⟨E-STMT⟩ `else` ⟨E-STMT⟩
⟨IF-THEN-ELSE⟩ → `if conditon then` ⟨E-STMT⟩ `else` ⟨STMT⟩

**2.22** $L(G)$ is the set of strings of 0s and #s that either contain exactly two #s and any number of 0s, or contain exactly one # and the number of 0s to the right of the # is twice the number of 0s to the left. First we show that three is not a pumping length for this language. The string 0#00 has length at least three, and it is in $L(G)$. It cannot be pumped using $p = 3$, because the only way to divide it into $uvxyz$ satisfying the first two conditions of the pumping lemma is $u = z = \varepsilon$, $v = 0$, $x = \#$, and $y = 00$, but that division fails to satisfy the third condition.

Next, we show that 4 is a pumping length for $L(G)$. If $w \in L(G)$ has length at least 4 and if it contains two #s, then it contains at least one 0. Therefore, by cutting $w$ into $uvxyz$ where either $v$ or $y$ is the string 0, we obtain a way to pump $w$. If $w \in L(G)$ has length at least 4 and if it contains a single #, then it must be of the form $0^k\#0^{2k}$ for some $k \geq 1$. Hence, by assigning $u = 0^{k-1}$, $v = 0$, $x = \#$, $y = 00$, and $z = 0^{2k-2}$, we satisfy all three conditions of the lemma.

**2.23** Let $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^l \mid i, j, k, l \geq 0 \text{ and if } i = 1 \text{ then } j = k = l\}$. $F$ is not context free because $F \cap \mathtt{ab}^*\mathtt{c}^*\mathtt{d}^* = \{\mathtt{ab}^n\mathtt{c}^n\mathtt{d}^n \mid n \geq 0\}$, which is not a CFL by the pumping lemma, and the intersection of a CFL and a regular language is a CFL (see Problem 2.17). However, $F$ does satisfy the pumping lemma with $p = 2$. ($p = 1$ works, too, with a bit more effort.)

If $s \in F$ has length 2 or more then:

i) If $s \in \mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ write $s$ as $s = rgt$ for $g \in \{b, c, d\}$ and divide $s = uvxyz$ where $u = r$, $v = g$, $x = y = \varepsilon$, $z = t$.

ii) If $s \in \mathtt{ab}^*\mathtt{c}^*\mathtt{d}^*$ write $s$ as $s = \mathtt{a}t$ and divide $s = uvxyz$ where $u = \varepsilon$, $v = \mathtt{a}$, $x = y = \varepsilon$, $z = t$.

iii) If $s \in \mathtt{aaa}^*\mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ write $s$ as $s = \mathtt{aa}t$ and divide $s = uvxyz$ where $u = \varepsilon$, $v = \mathtt{aa}$, $x = y = \varepsilon$, $z = t$.

In each of the three cases, we may easily check that the division of $s$ satisfies the conditions of the pumping lemma.

**2.24** Let $G$ be a CFG generating $A$. Let $p$ be the pumping length for $G$ given by the pumping lemma for CFGs. Let $k = p! = p(p-1)(p-2)\cdots 1$. Let $s = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$. We show that $s$ has two distinct parse trees in $G$. To do so we'll also consider the following two strings in $A$, $s_1 = \mathtt{a}^k\mathtt{b}^p\mathtt{c}^p$ and $s_2 = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^k$ with parse trees $\tau_1$ and $\tau_2$.

In $\tau_1$, remove all nodes that have only a's below them. The remaining subtree has $2p$ leaves that are only b's and c's. Because $2p > p$ we know that the subtree contains a path with a repeated has variable $R$, using the same argument that appears in the proof of the pumping lemma. Again by the same argument, use $R$ to divide $s$ into $s = uvxyz$, and $R$ can be chosen so that $|vxy| \leq p$. Strings $v$ and $y$ can each only contains one type of symbol; otherwise $uv^2xy^2z$ is not in $A$. Furthermore, $y$ contains no a's, because $R$

was on a path to a b or c. Hence $v$ must be b's and $y$ must be c's and they must have an equal length $l$. Letting $d = k/l$ (which must be an integer because $l \leq p$ and $k = p!$) the string $vu^dxy^d = s$ has a parse tree where most b's have a parent node in common with c's but not with a's. Repeating this argument with $\tau_2$ we obtain a parse tree for $s$ where most b's have a parent node in common with a's but not with c's. Hence $s$ has two distinct parse trees, so $G$ is ambiguous.

**2.25** The grammar generates all strings not of the form $\mathtt{a}^k\mathtt{b}^k$ for $k \geq 0$. Thus the complement of the language generated is $\overline{L(G)} = \{\mathtt{a}^k\mathtt{b}^k \mid k \geq 0\}$. The following grammar generates $\overline{L(G)}$: $\{\{S\}, \{\mathtt{a}, \mathtt{b}\}, \{S \rightarrow \mathtt{a}S\mathtt{b} \mid \epsilon\}, S\}$.

**2.26** Let $C = \{x\#y \mid x, y \in \{0,1\}^* \text{ and } x \neq y\}$. We construct a PDA $P$ that recognizes $C$. It operates by guessing corresponding positions on which the strings $x$ and $y$ differ, as follows. It reads the input at the same time as it pushes some symbols, say 1s, onto the stack. At some point it nondeterministically guesses a position in $x$ and it records the symbol it is currently reading there in its finite memory and skips to the #. Then it pops the stack while reading symbols from the input until the stack is empty and checks that the symbol it is now currently reading is different from the symbol it had recorded. If so, it accepts.

Here is a more detailed description of $P$'s algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, $P$ rejects on that branch of the computation.

1. Read next input symbol, and push 1 onto stack.
2. Nondeterministically jump to either 1 or 3.
3. Record the current input symbol in the finite control. For convenience call this symbol $a$.
4. Read input symbols until # is read.
5. Read the next symbol, and pop the stack.
6. If stack is empty, go to 7, otherwise go to 5.
7. *Accept* if the current input symbol differs from $a$, otherwise *reject*.

**2.27** Let $D = \{xy \mid x, y \in \{0,1\}^* \text{ and } |x| = |y| \text{ but } x \neq y\}$. We construct a PDA $P$ recognizing $D$. This PDA guesses corresponding places on which $x$ and $y$ differ. Checking that the places correspond is tricky. Doing so relies on the observation that the two corresponding places are $n/2$ symbols apart, where $n$ is the length of the entire input. Hence, by ensuring that the number of symbols between the guessed places is equal to the number other symbols, the PDA can check that the guessed places do indeed correspond,

Here we give a more detailed description of the PDA algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, $P$ rejects on that branch of the computation.

1. Read next input symbol and push 1 onto the stack.
2. Nondeterministically jump to either 1 or 3.
3. Record the current input symbol in the finite control. For convenience call this symbol $a$.
4. Read next input symbol and pop the stack. Repeat until stack is empty.
5. Read next input symbol and push 1 onto the stack.
6. Nondeterministically jump to either 5 or 7.
7. *Reject* if current input symbol differs from $a$.
8. Read next input symbol and pop the stack. Repeat until stack is empty.
9. *Accept* if input is empty.

Alternatively we can give a CFG for this language as follows.

$$\begin{aligned} S &\rightarrow AB \mid BA \\ A &\rightarrow XAX \mid 0 \\ B &\rightarrow XBX \mid 1 \\ X &\rightarrow 0 \mid 1 \end{aligned}$$

**2.28** Let $G = (\Sigma, V, T, P, S)$ be a context free grammar for $A$. Define $r$ to be the length of the longest string of symbols occurring on the right hand side of any production in $G$. We set the pumping length $k$ to be $r^{2|V|+1}$. Let $s$ be any string in $A$ of length at least $k$ and let $T$ be a derivation tree for $A$ with the fewest number of nodes. Observe that since $s \geq k$, the depth of $T$ must be at least $2|V|+1$. Thus, some path $p$ in $T$ has length at least $2|V|+1$. By the pigeonhole principle, some variable $V'$ appears at least three times in $p$. Label the last three occurrences of $V'$ in $p$ as $V_1$, $V_2$, and $V_3$. Moreover, define the strings generated by $V_1$, $V_2$, and $V_3$ in $T$ as $s_1$, $s_2$, and $s_3$ respectively. Now, each of these strings is nested in the previous because they are all being generated on the same path. Suppose then that $s_1 = l_1 s_2 r_1$ and $s_2 = l_2 s_3 r_2$. We now have three cases to consider:

i) $|l_1 l_2| \geq 1$ and $|r_1 r_2| \geq 1$: In this case, there is nothing else to prove since we can simply pump in the usual way.

ii) $|l_1 l_2| = 0$: Since we defined $T$ to be a minimal tree, neither $r_1$ nor $r_2$ can be the empty string. So, we can now pump $r_1$ and $r_2$.

iii) $|r_1 r_2| = 0$: This case is handled like the previous one.

# Chapter 3

**3.1**

a. $q_1 0, \sqcup q_2 \sqcup, \sqcup\sqcup q_{\text{accept}}$

b. $q_1 00, \sqcup q_2 0, \sqcup \mathbf{x} q_3 \sqcup, \sqcup q_5 \mathbf{x} \sqcup, q_5 \sqcup \mathbf{x} \sqcup, \sqcup q_2 \mathbf{x} \sqcup, \sqcup \mathbf{x} q_2 \sqcup, \sqcup \mathbf{x} \sqcup q_{\text{accept}}$

c. $q_1 000, \sqcup q_2 00, \sqcup \mathbf{x} q_3 0, \sqcup \mathbf{x} 0 q_4 \sqcup, \sqcup \mathbf{x} 0 \sqcup q_{\text{reject}}$

d. $q_1 000000, \sqcup q_2 00000, \sqcup x q_3 0000 \sqcup, \sqcup x 0 q_4 000, \sqcup x 0 x q_3 00, \sqcup x 0 x 0 q_4 0,$
$\sqcup x 0 x 0 x q_3 \sqcup, \sqcup x 0 x 0 q_5 x \sqcup, \sqcup x 0 x q_5 0 x \sqcup, \sqcup x 0 q_5 x 0 x \sqcup, \sqcup x q_5 0 x 0 x \sqcup, \sqcup q_5 x 0 x 0 x \sqcup,$
$q_5 \sqcup x 0 x 0 x \sqcup, \sqcup q_2 x 0 x 0 x \sqcup, \sqcup x q_2 0 x 0 x \sqcup, \sqcup x x q_3 x 0 x \sqcup, \sqcup x x x q_3 0 x \sqcup, \sqcup x x x 0 q_4 x \sqcup,$
$\sqcup x x x 0 x q_4 \sqcup, \sqcup x x x 0 x \sqcup q_{\text{reject}}$

**3.2**

a. $q_1 11, \sqcup q_3 1, \sqcup 1 q_3 \sqcup, \sqcup 1 \sqcup q_{\text{reject}}$

b. $q_1 1\#1, \sqcup q_3 \#1, \sqcup \# q_5 1, \sqcup \# 1 q_5 \sqcup, \sqcup \# q_7 1 \sqcup, \sqcup q_7 \# 1 \sqcup, q_7 \sqcup \# 1 \sqcup, \sqcup q_9 \# 1 \sqcup, \sqcup \# q_{11} 1 \sqcup,$
$\sqcup q_{12} \# x \sqcup, q_{12} \sqcup \# x \sqcup, \sqcup q_{13} \# x \sqcup, \sqcup \# q_{14} x \sqcup, \sqcup \# x q_{14} \sqcup, \sqcup \# x \sqcup q_{\text{accept}}$

c. $q_1 1\#\#1, \sqcup q_3 \#\#1, \sqcup \# q_5 \#1, \sqcup \#\# q_{\text{reject}} 1$

d. $q_1 10\#11, \sqcup q_3 0\#11, \sqcup 0 q_3 \#11, \sqcup 0 \# q_5 11, \sqcup 0 \# 1 q_5 1, \sqcup 0 \# 11 q_5 \sqcup, \sqcup 0 \# 1 q_7 1 \sqcup,$
$\sqcup 0 \# q_7 11 \sqcup, \sqcup 0 q_7 \# 11 \sqcup, \sqcup q_7 0 \# 11 \sqcup, q_7 \sqcup 0 \# 11 \sqcup, \sqcup q_9 0 \# 11 \sqcup, \sqcup 0 q_9 \# 11 \sqcup, \sqcup 0 \# q_{11} 11 \sqcup,$
$\sqcup 0 q_{12} \# x 1 \sqcup, \sqcup q_{12} 0 \# x 1 \sqcup, q_{12} \sqcup 0 \# x 1 \sqcup, \sqcup q_{13} 0 \# x 1 \sqcup, \sqcup x q_8 \# x 1 \sqcup, \sqcup x \# q_{10} x 1 \sqcup,$
$\sqcup x \# x q_{10} 1 \sqcup, \sqcup x \# x 1 q_{\text{reject}} \sqcup$

e. $q_1 10\#10, \sqcup q_3 0\#10, \sqcup 0 q_3 \#10, \sqcup 0 \# q_5 10, \sqcup 0 \# 1 q_5 0, \sqcup 0 \# 10 q_5 \sqcup, \sqcup 0 \# 1 q_7 0 \sqcup,$
$\sqcup 0 \# q_7 10 \sqcup, \sqcup 0 q_7 \# 10 \sqcup, \sqcup q_7 0 \# 10 \sqcup, q_7 \sqcup 0 \# 10 \sqcup, \sqcup q_9 0 \# 10 \sqcup, \sqcup 0 q_9 \# 10 \sqcup, \sqcup 0 \# q_{11} 10 \sqcup,$
$\sqcup 0 q_{12} \# x 0 \sqcup, \sqcup q_{12} 0 \# x 0 \sqcup, q_{12} \sqcup 0 \# x 0 \sqcup, \sqcup q_{13} 0 \# x 0 \sqcup, \sqcup x q_8 \# x 0 \sqcup, \sqcup x \# q_{10} x 0 \sqcup,$
$\sqcup x \# x q_{10} 0 \sqcup, \sqcup x \# q_{12} x x \sqcup, \sqcup x q_{12} \# x x \sqcup, \sqcup q_{12} x \# x x \sqcup, q_{12} \sqcup x \# x x \sqcup, \sqcup q_{13} x \# x x \sqcup,$
$\sqcup x q_{13} \# x x \sqcup, \sqcup x \# q_{14} x x \sqcup, \sqcup x \# x q_{14} x \sqcup, \sqcup x \# x x q_{14} \sqcup, \sqcup x \# x x \sqcup q_{\text{accept}}$

**3.3** If a language $L$ is decidable, it can be decided by a deterministic TM and that is automatically a nondeterministic TM.

If a language $L$ is decided by a nondeterministic TM $N$, we construct a deterministic TM $D_2$ that decides $L$. TM $D_2$ uses the same algorithm as in the TM $D$ described in the proof of Theorem 3.10, with an additional step 5: *Reject* if all branches of nondeterminism of $N$ are exhausted.

We argue that $D_2$ is a decider for $L$. If $N$ accepts, $D_2$ will eventually find an accepting branch and accept, too. If $N$ rejects, all of its branches halt and reject. Therefore, by the theorem on trees given in the exercise, its entire computation tree is finite, and $D$ will halt and reject when this entire tree has been explored.

**3.4** An enumerator is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{print}}, q_{\text{accept}})$, where $Q, \Sigma, \Gamma$ are all finite sets and

i) $Q$ is the set of states,
ii) $\Gamma$ is the work tape alphabet,
iii) $\Sigma$ is the output tape alphabet,
iv) $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\} \times \Sigma_\varepsilon$ is the transition function,
v) $q_0 \in Q$ is the start state,
vi) $q_{\text{print}} \in Q$ is the print state, and
vii) $q_{\text{accept}} \in Q$ is the reject state, where $q_{\text{print}} \neq q_{\text{reject}}$.

The computation of an enumerator $E$ is defined as in an ordinary TM, except for the following points. It has two tapes, a work tape and a print tape, both initially blank. At each step, the machine may write a symbol from $\Sigma$ on the output tape, or nothing, as determined by $\delta$. If $\delta(q, a) = (r, b, \mathrm{L}, c)$, it means that in state $q$, reading $a$, enumerator $E$ enters state $r$, writes $b$ on the work tape, moves the work tape head left (or right, if L had been R), writes $c$ on the output tape, and moves the output tape head to the right if$c \neq \varepsilon$.

Whenever state $q_{\text{print}}$ is entered, the output tape is reset to blank and the head returns to the left-hand end. The machine halts when $q_{\text{accept}}$ is entered. $L(E) = \{w \in \Sigma^* |\ w$ appears on the work tape if $q_{\text{print}}$ is entered$\}$.

**3.5**

**a.** Yes. $\Gamma$ is the tape alphabet. A Turing Machine can write any characters in $\Gamma$ on its tape, and $\sqcup \in \Gamma$ according to the definition.

**b.** No. $\Sigma$ never contains $\sqcup$ but $\Gamma$ always contains $\sqcup$. So, they cannot be equal.

**c.** Yes. If the Turing Machine tries to move its head off the left-hand end of the tape, it remains on the same tape cell.

**d.** No. Any Turing Machine must contain two distinct states $q_{\text{accept}}$ and $q_{\text{reject}}$. So, a Turing Machine contains at least two states.

**3.6** In Stage 2 of this algorithm: "Run $M$ on $s_i$", if $M$ loops on a certain input $s_i$, $E$ would not check any inputs after $s_i$. If that were to occur, $E$ might fail to enumerate $L(M)$ as required.

**3.7** The variables $x_1, \ldots, x_k$ have infinitely may possible settings. A Turing machine would required infinite time to try them all. But, we require that every stage in the Turing machine description be completed in a finite number of steps.

**3.8**

**a.** "On input string $w$:

1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4. Otherwise, move the head back to the front of the tape.
2. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *reject*.
3. Move the head back to the front of the tape and repeat stage 1.
4. Move the head back to the front of the tape. Scan the tape to see if any unmarked 1s remain. If there are none, *accept*. Otherwise, *reject*."

**b.** "On input string $w$:

1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4.

2. Move on and mark the next unmarked 0. If there is not any on the tape, *reject*. Otherwise, move the head back to the front of the tape.
3. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *reject*.
4. Move the head back to the front of the tape and repeat stage 1.
5. Move the head back to the front of the tape. Scan the tape to see if there are any unmarked 1s. If there is not, *accept*. Otherwise, *reject*."

c. "On input string $w$:
   1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4.
   2. Move on and mark the next unmarked 0. If there is not any on the tape, *accept*. Otherwise, move the head back to the front of the tape.
   3. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *accept*.
   4. Move the head back to the front of the tape and repeat stage 1.
   5. Move the head back to the front of the tape. Scan the tape to see if there are any unmarked 1s. If there is not, *reject*. Otherwise, *accept*."

**3.9** a. By Example 2.20, no PDA recognizes $B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. The following 2-PDA recognizes $B$. Push all the a's that appear in the front of the input tape to stack 1. Then push all the b's that follow the a's in the input stream into stack 2. If it sees any a's at this stage, *reject*. When it sees the first c, pop stack 1 and stack 2 at the same time. After this, reject the input if it sees any a's or b's in the input stream. Pop stack 1 and stack 2 for each input character c it reads. If the input ends when both stacks are empty, *accept*. Otherwise, *reject*.

b. We show that a 2-PDA can simulate a TM. Furthermore, a 3-tape NTM can simulate a 3-PDA, and an ordinary deterministic, 1-tape TM can simulate a 3-tape NTM. Therefore a 2-PDA can simulate a 3-PDA, and so they are equivalent in power.

The TM by 2-PDA simulation is done as follows. We split the tape of a TM into two stacks. Stack 1 stores the characters on the left of the head, with the bottom of stack storing the leftmost character of the tape in the TM. Stack 2 stores the characters on the right of the head, with the bottom of the stack storing the rightmost character on the tape in the TM. In other words, if a TM configuration is $aq_ib$, the corresponding 2-PDA is in state $q_i$, with stack 1 storing the string $a$ and stack 2 storing the string $b^R$, both from bottom to top.

For each transition $\delta(q_i, c_i) = (q_j, c_j, \mathrm{L})$ in the TM, the corresponding PDA transition pops $c_i$ off stack 2, pushes $c_j$ into stack 2, pops stack 1 and pushes the character into stack 2, and goes from state $q_i$ to $q_j$. For any transition $\delta(q_i, c_i) = (q_j, c_j, \mathrm{R})$ in the TM, the corresponding PDA transition pops $c_i$ off stack 2 and takes away $c_i$, push $c_j$ into stack 1, and goes from state $q_i$ to $q_j$.

**3.10** We first simulate an ordinary TM by a write-twice TM. The write-twice TM simulates a single step of the original TM by copying the entire tape over to a fresh section of the tape to the right of the currently used section. The copying procedure operates character by character, marking a character as it is copied. This procedure alters each tape square twice, once to write the character for the first time and another time to mark that it is copied. The position of the original TMs tape head is marked on the tape. When copying the cells around the the marked position, the tape contents is updated according to the rules of the TM.

To carry out the simulation with a write-once TM, operate as before, except that each cell of the tape is represented by two cells. The first of these contains the original TM's tape contents and the second is for the mark used in the copying procedure. The input is not presented to the machine in the format with two cells per symbol, so the very first time the tape is copied, the copying marks are put directly over the input symbols.

**3.11** A Turing machine with doubly infinite tape can easily simulate an ordinary Turing Machine. It needs to mark the left-hand end of the input so that it can prevent the head from moving off of that end.

To simulate the doubly infinite tape TM by an ordinary TM, we show how to simulate it with a 2-tape TM, which was already shown to be equivalent in power to an ordinary TM. The first tape of the 2-tape TM is written with the input string and the second tape is blank. We cut the tape of the doubly infinite tape TM into two parts, at the starting cell of the input string. The portion with the input string and all the blank spaces to its right appears on the first tape of the 2-tape TM. The portion to the left of the input string appears on the second tape, in reverse order.

**3.12** We simulate an ordinary TM with a reset TM has only the RESET and R operations. When the ordinary TM moves its head right, the reset TM does the same. When the ordinary TM moves its head left, the reset TM cannot, so it gets the same effect by marking the current head location on the tape, then resetting and copying the entire tape one cell to the right, except for the mark, which is kept on the same tape cell. Then it resets again, and scans right until it find the mark.

**3.13** We simulate this TM variant by an NFA. The NFA only needs to keep track of the TM's current state and what the TMs written on the current tape cell. Remembering what it has written on tape cells to the left of the current head position is unnecessary, because the TM is unable to return to these cells an read them. Using an NFA in the actual construction is convenient because it allows $\varepsilon$ moves which are useful for simulating the "stay put" TM transitions.

More formally, the NFA has $|Q \times \Gamma| + 3$ states. The first $|Q \times \Gamma|$ states correspond to the state of the TM variant and the character on the tape that the head of the TM points to. The extra states are the start state $q_{\text{start}}$, a special accept state $q_{\text{accept}}$ and a special reject state $q_{\text{reject}}$. The transition function $\delta'$ for the NFA is constructed according $\delta$. First we set $\delta'(q_{\text{start}}, p) = \{q_{0p}\}$, where $q_0$ is the start state of the TM variant. Next, we set $\delta'(q_{\text{accept}}, i) = \{q_{\text{accept}}\}$ for any $i$.

If $\delta(p, a) = (q_{\text{accept}}, b, w)$, where $w =$ R or S, we set $\delta'(q_{pa}, \varepsilon) = \{q_{\text{accept}}\}$. If $\delta(p, a) = (q_{\text{reject}}, b, w)$, where $w =$ R or S, we set $\delta'(q_{pa}, \varepsilon) = \{q_{\text{reject}}\}$.

**3.14** **a.** For any two decidable languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the TMs that decide them. We construct a TM $M'$ that decides the union of $L_1$ and $L_2$:

"On input $w$:

1. Run $M_1$ on $w$, if it accepts, *accept*.
2. Run $M_2$ on $w$, if it accepts, *accept*. Otherwise, *reject*."

$M'$ accepts $w$ if either $M_1$ or $M_2$ accepts it. If both reject, $M'$ rejects.

**b.** For any two decidable languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the TMs that decide them. We construct a NTM $M'$ that decides the concatenation of $L_1$ and $L_2$:

"On input $w$:

1. For each way to cut $w$ into two parts $w = w_1 w_2$:
2. Run $M_1$ on $w_1$.
3. Run $M_2$ on $w_2$.
4. If both accept, *accept*. Otherwise, continue with the next $w_1, w_2$.
5. All cuts have been tried without success, so *reject*."

We try every possible cut of $w$. If we ever come across a cut such that the first part is accepted by $M_1$ and the second part is accepted by $M_2$, $w$ is in the concatenation of $L_1$ and $L_2$. So $M'$ accept $w$. Otherwise, $w$ does not belong to the concatenation of the languages and is rejected.

**c.** For any decidable language $L$, let $M$ be the TM that decides it. We construct a NTM $M'$ that decides the star of $L$:

"On input $w$:

1. For each way to cut $w$ into parts so that $w = w_1 w_2 \ldots w_n$:
2. Run $M$ on $w_i$ for $i = 1, 2, \ldots, n$. If $M$ accepts each of these string $w_i$, *accept*.
3. All cuts have been tried without success, so *reject*."

If there is a way to cut $w$ into different substrings such that every substring is accepted by $M$, $w$ belongs to the star of $L$ and thus $M'$ accepts $w$. Otherwise, $w$ is rejected. Since there are finitely many possible cuts of $w$, $M'$ will halt after finitely many steps.

**d.** For any decidable language $L$, let $M$ be the TM that decides it. We construct a TM $M'$ that decides the complement of $L$:

"On input $w$:

1. Run $M$ on $w$. If $M$ accepts, *reject*; if $M$ rejects, *accept*."

Since $M'$ does the opposite of whatever $M$ does, it decides the complement of $L$.

**e.** For any two decidable languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the TMs that decide them. We construct a TM $M'$ that decides the intersection of $L_1$ and $L_2$:

"On input $w$:

1. Run $M_1$ on $w$, if it rejects, *reject*.
2. Run $M_2$ on $w$, if it accepts, *accept*. Otherwise, *reject*."

$M'$ accepts $w$ if both $M_1$ and $M_2$ accept it. If either of them rejects, $M'$ rejects $w$, too.

**3.15** **a.** For any two Turing-recognizable languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the TMs that recognize them. We construct a TM $M'$ that recognizes the union of $L_1$ and $L_2$:

"On input $w$:

1. Run $M_1$ and $M_2$ alternatively on $w$ step by step. If either accept, *accept*. If both halt and reject, then *reject*."

If any of $M_1$ and $M_2$ accept $w$, $M'$ will accept $w$ since the accepting TM will come to its accepting state after a finite number of steps. Note that if both $M_1$ and $M_2$ reject and either of them does so by looping, then $M$ will loop.

**b.** For any two Turing-recognizable languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the TMs that recognize them. We construct a NTM $M'$ that recognizes the concatenation of $L_1$ and $L_2$:

"On input $w$:

1. Nondeterministically cut $w$ into two parts $w = w_1 w_2$.
2. Run $M_1$ on $w_1$. If it halts and rejects, *reject*. If it accepts, go to stage 3.
3. Run $M_2$ on $w_2$. If it accepts, *accept*. If it halts and rejects, *reject*."

If there is a way to cut $w$ into two substrings such $M_1$ accepts the first part and $M_2$ accepts the second part, $w$ belongs to the concatenation of $L_1$ and $L_2$ and $M'$ will accept $w$ after a finite number of steps.

**c.** For any Turing-recognizable language $L$, let $M$ be the TM that recognizes it. We construct a NTM $M'$ that recognizes the star of $L$:

"On input $w$:

1. Nondeterministically cut $w$ into parts so that $w = w_1 w_2 \cdots w_n$.
2. Run $M$ on $w_i$ for all $i$. If $M$ accepts all of them, *accept*. If it halts and rejects any of them, *reject*."

If there is a way to cut $w$ into substrings such $M$ accepts the all the substrings, $w$ belongs to the star of $L$ and $M'$ will accept $w$ after a finite number of steps.

**d.** For any two Turing-recognizable languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the TMs that recognize them. We construct a TM $M'$ that recognizes the intersection of $L_1$ and $L_2$:

"On input $w$:

1. Run $M_1$ on $w$. If it halts and rejects, *reject*. If it accepts, go to stage 3.
2. Run $M_2$ on $w$. If it halts and rejects, *reject*. If it accepts, *accept*."

If both of $M_1$ and $M_2$ accept $w$, $w$ belongs to the intersection of $L_1$ and $L_2$ and $M'$ will accept $w$ after a finite number of steps.

**3.16** If $A$ is decidable, the enumerator operates by generating the strings in lexicographic order and testing each in turn for membership in $A$ using the decider. Those strings which are found to be in $A$ are printed.

If $A$ is enumerable in lexicographic order, we consider two cases. If $A$ is finite, it is decidable because all finite languages are decidable. If $A$ is infinite, a decider for $A$ operates as follows. On receiving input $w$, the decider enumerates all strings in $A$ in order until some string lexicographically after $w$ appears. That must occur eventually because $A$ is infinite. If $w$ has appeared in the enumeration already then *accept*, but if it hasn't appeared yet, it never will, so *reject*.

Note: Breaking into two cases is necessary to handle the possibility that the enumerator may loop without producing additional output when it is enumerating a finite language. As a result, we end up showing that the language is decidable but we do not (and cannot) algorithmically construct the decider for the language from the enumerator for the language. This subtle point is the reason for the star on the problem.

**3.17** We give a DFA $A$ that is equivalent to TM $M$. Design $A$ in two stages. First, we arrange the states and transitions of $A$ to simulate $M$ on the read-only input, portion of the tape. Second, we assign the accept states of $A$ to correspond to the action of $M$ on the read/write portion of the tape to the right of the input. In the first stage, we construct $A$ to store a function

$$F_s\colon (Q \cup \{\text{first}\}) \longrightarrow (Q \cup \{\texttt{acc}, \texttt{rej}\})$$

in its finite control where $Q$ is the set of $M$'s states and $s$ is the string that it has read so far. Note that only finitely many such functions exist, so we assign one state for each possibility.

The function $F_s$ contains information about the way $M$ would compute on a string $s$. In particular, $F_s(\text{first})$ is the state that $M$ enters when it is about to move off of the right end of $s$ for the first time, when $M$ is started in its starting state at the left end of $s$. If $M$ accepts or rejects (either explicitly or by looping) before ever moving off of the right end of $s$, then $F_s(\text{first}) = \texttt{acc}$ or $\texttt{rej}$ accordingly. Furthermore, for $F_s(q)$ is the state $M$ enters when it is about to move off of the right end of $s$ for the first time, when $M$ is started in state $q$ at the right end of $s$. As before, if $M$ accepts or rejects before ever moving off of the right end of $s$, then $F_s(q) = \texttt{acc}$ or $\texttt{rej}$.

Observe that, for any string $s$ and symbol $a$ in $\Sigma$, we can determine $F_{sa}$ from $a$, $F_s$, and $M$'s transition function. Furthermore, $F_\varepsilon$ is predetermined because $F_\varepsilon(\text{first}) = q_0$, the start state of $M$, and $F_\varepsilon(q) = q$ (because TMs that attempt to move off the leftmost end of the tape just stay in the left-hand cell). Hence we can obtain $A$'s start state and transition function. That completes the first stage of $A$'s design.

To complete the second stage, we assign the accept states of $M$. For any two strings $s$ and $t$, if $F_s = F_t$, then $M$ must have the same output on $s$ and $t$—either both $s$ and $t$ are in $L(M)$ or both are out. Hence $A$'s state at the end of $s$ is enough to determine whether $M$ accepts $s$. More precisely, we declare a state of $A$ to be accepting if it was assigned to a function $F$ that equals $F_s$ for any string $s$ that $M$ accepts.

**3.19** The language $A$ is one of the two languages, $\{0\}$ or $\{1\}$. In either case the language is finite, and hence decidable. If we are not able to determine which of these two languages is $A$, we won't be able to describe the decider for $A$, but we can give two TMs, one of which is $A$'s decider.

# Chapter 4

**4.1**
a. Yes
b. No
c. No
d. No
e. No
f. Yes

**4.2** Let $EQ_{\mathsf{DFA,REX}} = \{\langle A, R\rangle \mid A \text{ is a DFA}, R \text{ is a regular expression and } L(A) = L(R)\}$. The following TM $E$ decides $EQ_{\mathsf{DFA,REX}}$.

$E$ = "On input $\langle A, R\rangle$:

1. Convert regular expression $R$ to an equivalent DFA $B$ using the procedure given in Theorem 1.28.
2. Use the TM $C$ for deciding $EQ_{\mathsf{DFA}}$ given in Theorem 4.5, on input $\langle A, B\rangle$.

3. If $R$ accepts, *accept*. If $R$ rejects, *reject*."

**4.3** Let $ALL_{\mathsf{DFA}} = \{\langle A\rangle |\ A$ is a DFA that recognizes $\Sigma^*\}$. The following TM $L$ decides $ALL_{\mathsf{DFA}}$.

$L =$ "On input $\langle A\rangle$ where $A$ is a DFA:

1. Construct DFA B that recognizes $\overline{L(A)}$ as described in Exercise 1.10.
2. Run TM $T$ from Theorem 4.4 on input $\langle B\rangle$, where $T$ decides $E_{\mathsf{DFA}}$.
3. If $T$ accepts, *accept*. If $T$ rejects, *reject*."

**4.4** Let $A\varepsilon_{\mathsf{CFG}} = \{\langle G\rangle |\ G$ is a CFG that generates $\varepsilon\}$. The following TM $V$ decides $A\varepsilon_{\mathsf{CFG}}$.

$V =$ "On input $\langle G\rangle$ where $G$ is a CFG:

1. Run TM $S$ from Theorem 4.6 on input $\langle G, \varepsilon\rangle$, where $S$ is a decider for $A_{\mathsf{CFG}}$.
2. If $S$ accepts, *accept*. If $S$ rejects, *reject*."

**4.5** Let $INFINITE_{\mathsf{DFA}} = \{\langle A\rangle |\ L(A)$ is an infinite language$\}$. The following TM $I$ decides $INFINITE_{\mathsf{DFA}}$.

$I =$ "On input $\langle A\rangle$ where $A$ is a DFA:

1. Let $k$ be the number of states of $A$.
2. Construct a DFA $D$ that accepts strings of length $\geq k$.
3. Construct a DFA $M$ such that $L(M) = L(A) \cap L(D)$.
4. Run TM $T$ from Theorem 4.4 on input $\langle M\rangle$, where $T$ decides $E_{\mathsf{DFA}}$.
5. If $T$ accepts, *reject*. If $T$ rejects, *accept*."

If $A$ accepts a string with length at least $k$, this string can be must cause $A$ to enter the same state at least twice, so it can be pumped up to obtain infinitely many strings that are accepted by $A$. Conversely, if $A$ accepts infinitely many strings, it must accept strings of arbitrarily long lengths, in particular, a string of length at least $k$.

**4.6**

a. $g$ is one-to-one. $f$ is not one-to-one because $f(1) = f(3)$.
b. $g$ is onto. $f$ is not onto because there does not exist $x \in X$ such that $f(x) = 10$.
c. $g$ is a correspondence because $g$ is one-to-one and onto. $f$ is not a correspondence because $f$ is not one-to-one and onto.

**4.7** Suppose $\mathcal{B}$ is countable and a correspondence $f: \mathcal{N} \longrightarrow \mathcal{B}$ exists. We construct $x$ in $\mathcal{B}$ that is not paired with anything in $\mathcal{N}$. Let $x = x_1 x_2 \ldots$. Let $x_i = 0$ if $f(i)_i = 1$, and $x_i = 1$ if $f(i)_i = 0$ where $f(i)_i$ is the $i$th bit of $f(i)$. Therefore, we ensure that $x$ is not $f(i)$ for any $i$ because it differs from $f(i)$ in the $i$th symbol, and a contradiction occurs.

**4.8** We demonstrate a one-to-one $f: T \longrightarrow \mathcal{N}$. Let $f(i,j,k) = 2^i 3^j 5^k$. Function $f$ is one-to-one because if $a \neq b$, $f(a) \neq f(b)$. Therefore, $T$ is countable.

**4.9** Let $\sim$ be the binary relation "is the same size". In other words If $A \sim B$, $A$ and $B$ are the same size. We show that $\sim$ is an equivalence relation. First, $\sim$ is reflexive because the identity function $f(x) = x, \forall x \in A$ is a correspondence $f: A \longrightarrow A$. Second, $\sim$ is symmetric because any correspondence has an inverse, which itself is a correspondence. Third, $\sim$ is transitive because if $A \sim B$ via correspondence $f$, and $B \sim C$ via correspondence $g$, then $A \sim C$ via correspondence $f \circ g$ (the composition of $f$ and $g$). Because $\sim$ is reflexive, symmetric, and transitive, $\sim$ is an equivalence relation.

**4.10** The following TM $X$ decides $A$.

$X$ = "On input $\langle M \rangle$ where $M$ is a DFA:

1. Construct a DFA $O$ that accepts any string containing an odd number of 1s.
2. Construct DFA $B$ such that $L(B) = L(M) \cap L(O)$.
3. Run TM $T$ from Theorem 4.4 on input $\langle B \rangle$, where $T$ decides $E_{\mathsf{DFA}}$.
4. If $T$ accepts, *accept*. If $T$ rejects, *reject*."

**4.11** We observe that $L(R) \subseteq L(S)$ if and only if $\overline{L(S)} \cap L(R) = \emptyset$. The following TM $X$ decides $A$.

$X$ = "On input $\langle R, S \rangle$ where $R$ and $S$ are regular expressions:

1. Construct DFA $E$ such that $L(E) = \overline{L(S)} \cap L(R)$.
2. Run TM $T$ from Theorem 4.4 on input $\langle E \rangle$, where $T$ decides $E_{\mathsf{DFA}}$.
3. If $T$ accepts, *accept*. If $T$ rejects, *reject*."

**4.12** We have shown in Problem 2.17 that if $C$ is a context-free language and $R$ is a regular language, then $C \cap R$ is context free. Therefore $1^* \cap L(G)$ is context free. The following TM $X$ decides $A$.

$X$ = "On input $\langle G \rangle$ where $G$ is a CFG:

1. Construct CFG $H$ such that $L(H) = 1^* \cap L(G)$.
2. Run TM $R$ from Theorem 4.7 on input $\langle H \rangle$, where $R$ decides $E_{\mathsf{CFG}}$.

3. If $R$ accepts, *reject*. If $T$ rejects, *accept*."

**4.14** The following TM $X$ decides $A$.

$X$ = "On input $\langle R \rangle$ where $R$ is a regular expression:

1. Construct DFA $E$ that accepts $\Sigma^* 111 \Sigma^*$.
2. Construct DFA $B$ such that $L(B) = L(R) \cap L(E)$.
3. Run TM $T$ from Theorem 4.4 on input $\langle B \rangle$, where $T$ decides $E_{\mathsf{DFA}}$.
4. If $T$ accepts, *reject*. If $T$ rejects, *accept*."

**4.15** Let $A = \{v \mid v \text{ is of the form } ww^{\mathcal{R}} \text{ for } w \in \{0,1\}^*\}$. $A$ is a CFL. Problem 2.17 showed that $B \cap A$ is context free if $B$ is regular. The following TM $X$ decides $E$.

$X$ = "On input $\langle M \rangle$ where $M$ is a DFA:

1. Let $C = L(M) \cap A$. Let $G$ be the CFG of $C$.
2. Run TM $R$ from Theorem 4.7 on input $\langle G \rangle$, where $R$ decides $E_{\mathsf{CFG}}$.
3. If $R$ accepts, *reject*. If $R$ rejects, *accept*."

$X$ decides $E$ because a DFA $M$ accepts some string of the form $ww^{\mathcal{R}}$ if and only if $L(M) \cap A \neq \emptyset$.

**4.16** For any DFAs $A$ and $B$, $L(A) = L(B)$ if and only if $A$ and $B$ accept the same strings up to length $mn$, where $m$ and $n$ are the numbers of states of $A$ and $B$, respectively. Obviously, if $L(A) = L(B)$, $A$ and $B$ will accept the same strings. If $L(a) \neq L(B)$, we need to show that the languages differ on some string $s$ of length at most $mn$. Let $t$ be a shortest string on which the languages differ. Let $l$ be the length of $t$. If $l \leq mn$ we are done. If not, consider the sequence of states $q_0, q_1, \ldots, q_l$ that $A$ enters on input $t$, and the sequence of states $r_0, r_1, \ldots, r_l$ that $B$ enters on input $t$. Because $A$ has $m$ states and $B$ has $n$ states, only $mn$ distinct pairs $(q, r)$ exist where $q$ is a state of $A$ and $r$ is a state of $B$. By the pigeon hole principle, two pairs of states $(q_i, r_i)$ and $(q_j, r_j)$ must be identical. If we remove the portion of $t$ from $i$ to $j-1$ we obtain a shorter string on which $A$ and $B$ behave as they would on $t$. Hence we have found a shorter string on which the two languages differ, even though $t$ was supposed to be shortest. Hence $t$ must be no longer than $mn$.

**4.17** We need to prove both directions. To handle the easier one first, assume that $D$ exists. A TM recognizing $C$ operates on input $x$ by going through each possible string $y$ and testing whether $\langle x, y \rangle \in D$. If such a $y$ is ever found, *accept*; if not, just continue searching.

For the other direction, assume that $C$ is recognized by TM $M$. Define a language $B$ to be $\{\langle x, y\rangle |\ M$ accepts $x$ within $|y|$ steps$\}$. Language $B$ is decidable, and if $x \in C$ then $M$ accepts $x$ within some number of steps, so $\langle x, y\rangle \in B$ for any sufficiently long $y$, but if $x \notin C$ then $\langle x, y\rangle \notin C$ for any $y$.

**4.18** Let $A$ and $B$ be two languages such that $A \cap B = \emptyset$, and $\overline{A}$ and $\overline{B}$ are recognizable. Let $J$ be the TM recognizing $\overline{A}$ and $K$ be the TM recognizing $\overline{B}$. We will show that the language decided by TM $T$ separates $A$ and $B$.

$T$ = "On input $w$:

1. Simulate $J$ and $K$ on $w$ by alternating the steps of the two machines.
2. If $J$ accepts first, *reject*. If $K$ accepts first, *accept*."

The algorithm $T$ terminates because $\overline{A} \cup \overline{B} = \Sigma^*$. So either $J$ or $K$ will accept $w$ eventually. $A \subseteq C$ because if $w \in A$, $w$ will not be recognized by $J$ and will be accepted by $K$ first. $B \subseteq \overline{C}$ because if $w \in B$, $w$ will not be recognized by $K$ and will be accepted by $J$ first. Therefore, C separates $A$ and $B$.

**4.19** For any language $A$, let $A^{\mathcal{R}} = \{w^{\mathcal{R}} |\ w \in A\}$. If $\langle M\rangle \in S$, then $L(M) = L(M)^{\mathcal{R}}$. The following TM $T$ decides $S$.

$T$ = "On input $\langle M\rangle$, where $M$ is a DFA:

1. Construct DFA $N$ that recognizes $L(M)^{\mathcal{R}}$.
2. Run TM $F$ from Theorem 4.5 on $\langle M, N\rangle$, where $F$ is the Turing machine deciding $EQ_{\mathsf{DFA}}$.
3. If $F$ accepts, *accept*. If $F$ rejects, *reject*."

**4.20** Let $U = \{\langle P\rangle |\ P$ is a PDA that has useless states$\}$. The following TM $T$ decides $U$.

$T$ = "On input $\langle P\rangle$, where $U$ is a PDA:

1. For each state $q$ of $P$:
2. Modify $P$ so that $q$ is the only accept state.
3. Use the decider for $E_{\mathsf{PDA}}$ to test whether the modified PDA's language is empty. If it is, *accept*. If it is not, continue.
4. At this point, all states have been shown to be useful, so *reject*."

**4.21** Use the diagonalization method to obtain $D$. Let $\langle M_1\rangle, \langle M_2\rangle, \ldots$ be the output of an enumerator for $A$. Let $\Sigma = \{s_1, s_2, \ldots\}$ be a list of all strings over the alphabet. The algorithm for $D$ is:

$D$ = "On input $w$:

1. Let $i$ be the index of $w$ on the list of all strings, that is, $s_i = w$.
2. Run $\langle M_i\rangle$ on input $w$.

3. If it accepts, *reject*. If it rejects, *accept*."

$D$ is a decider because each $M_i$ is a decider. But $D$ doesn't appear on the enumeration for $A$, because it differs from $M_i$ on input $s_i$.

**4.22** Let $K$ be the enumerator for $B$. The following TM $L$ is the lexicographic-order enumerator for $C$.

$L$ = "Ignore the input.

1. Enumerate machines $M_i$ by $K$.
2. For each $M_i$, insert useless states, symbols or transitions to obtain $N_i$ such that $L(M_i) = L(N_i)$ and $\langle N_i \rangle$ is lexicographically larger than all $\langle N_j \rangle$ for $1 \leq j < i$. Output $N_i$."

Let $\langle N_1 \rangle, \langle N_2 \rangle \ldots$ be the strings enumerated by $L$. Since $L(N_i) = L(M_i)$ for all $i$, every machine described in $B$ has an equivalent one in $C$ and vice versa. $C$ is decidable because it enumerated by $L$ in lexicographic order, by virtue of Problem 3.16.

# Chapter 5

**5.1** Suppose for a contradiction that $EQ_{\mathsf{CFG}}$ were decidable. We construct a decider $M$ for $ALL_{\mathsf{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$ as follows:

$M$ = "On input $\langle G \rangle$:

1. Construct a CFG $H$ such that $L(H) = \Sigma^*$.
2. Run the decider for $EQ_{\mathsf{CFG}}$ on $\langle G, H \rangle$.
3. If it accepts, *accept*. If it rejects, *reject*."

$M$ decides $ALL_{\mathsf{CFG}}$ assuming a decider for $EQ_{\mathsf{CFG}}$ exists. Since we know $ALL_{\mathsf{CFG}}$ is undecidable, we have a contradiction.

**5.2** Here is a Turing Machine $M$ which recognizes the complement of $EQ_{\mathsf{CFG}}$:

$M$ = "On input $\langle G, H \rangle$:

1. Lexicographically generate the strings $x \in \Sigma^*$.
2. For each such string $x$:
3. Test whether $x \in L(G)$ and whether $x \in L(H)$, using the algorithm for $A_{\mathsf{CFG}}$.
4. If one of the tests accepts and the other rejects, *accept*; otherwise, continue."

**5.3** Here is a match: $\left[\frac{\mathtt{ab}}{\mathtt{abab}}\right]\left[\frac{\mathtt{ab}}{\mathtt{abab}}\right]\left[\frac{\mathtt{aba}}{\mathtt{b}}\right]\left[\frac{\mathtt{b}}{\mathtt{a}}\right]\left[\frac{\mathtt{b}}{\mathtt{a}}\right]\left[\frac{\mathtt{aa}}{\mathtt{a}}\right]\left[\frac{\mathtt{aa}}{\mathtt{a}}\right]$.

**5.4** No, it does not imply that $A$ is regular. For example, $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\} \leq_{\mathrm{m}} \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$. The reduction first tests whether its input is a member of $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. If so, it outputs the string ab, and if not, it outputs the string a.

**5.5** Suppose for a contradiction that $A_{\mathsf{TM}} \leq_{\mathrm{m}} E_{\mathsf{TM}}$ via reduction $f$. It follows from the definition of mapping reducibility that $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{E_{\mathsf{TM}}}$ via the same reduction function $f$. Observe that $\overline{E_{\mathsf{TM}}}$ is Turing-recognizable, but, $\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable, contradicting Theorem 5.22.

**5.6** Suppose $A \leq_{\mathrm{m}} B$ and $B \leq_{\mathrm{m}} C$. Then there are computable functions $f$ and $g$ such that $x \in A \iff f(x) \in B$ and $y \in B \iff g(y) \in C$. Consider the function composition $h(x) = g(f(x))$. We can build a TM which computes $h$ as follows: first simulate a TM for $f$ (such a TM exists since we assumed that $f$ is computable) on input $x$ and call the output $y$. Then simulate a TM for $g$ on $y$. The output is $h(x) = g(f(x))$. Therefore, $h$ is a computable function. Moreover, $x \in A \iff h(x) \in C$. Therefore, $A \leq_{\mathrm{m}} C$ via the reduction function $h$.

**5.7** Suppose $A \leq_{\mathrm{m}} \overline{A}$. Then, $\overline{A} \leq_{\mathrm{m}} A$, by the definition of mapping reducibility. Because $A$ is Turing-recognizable, Theorem 5.22 implies that $\overline{A}$ is Turing-recognizable, and then Theorem 4.16 implies that $A$ is decidable.

**5.8** We need to handle the case where the head is at the leftmost tape call and attempts to move left. To do so we add dominos

$$\left[\frac{\#qa}{\#rb}\right]$$

For every $q, r \in Q$ and $a, b \in \Gamma$, where $\delta(q, a) = (r, b, \mathrm{L})$.

**5.9** Suppose $L$ is a Turing-recognizable language and let $M$ be a TM that recognizes it. To reduce $L$ to $A_{\mathsf{TM}}$ we map any string $x$ to $\langle M, x\rangle$. Then $x \in L \iff \langle M, x\rangle \in A_{\mathsf{TM}}$. In addition, this mapping is computable, so it gives a mapping reduction from $L$ to $A_{\mathsf{TM}}$.

**5.10** To prove that $J$ is not Turing-recognizable, we show that $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} J$. The reduction function maps any string $y$ to the string $1y$. Then $y \in \overline{A_{\mathsf{TM}}} \iff 1y \in J$. This mapping is computable, so we have shown that $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} J$. To show that $\overline{J}$ is not Turing-recognizable we show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} J$ (since $A_{\mathsf{TM}} \leq_{\mathrm{m}} J \iff \overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{J}$). The reduction function maps any string $x$ to $0x$. Then $x \in A_{\mathsf{TM}} \iff 0x \in J$. This mapping is computable, so we have shown that $A_{\mathsf{TM}} \leq_{\mathrm{m}} J$.

**5.11** Use the language $J$ from Problem 5.10. That problem showed that $J$ is undecidable. Here we show that $J \leq_{\mathrm{m}} \overline{J}$. The reduction $f$ maps $\varepsilon$ to itself and for other strings $x$,

$$f(x) = \begin{cases} 0s & \text{if } x = 1s \text{ for some string } s \in \Sigma^* \\ 1s & \text{if } x = 0s \text{ for some string } s \in \Sigma^* \end{cases}$$

The function $f$ is computable, and $x \in J \iff f(x) \in \overline{J}$. Thus, $J \leq_{\mathrm{m}} \overline{J}$.

**5.12** We show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} S$ by mapping $\langle M, w\rangle$ to $\langle M'\rangle$ where $M'$ is the following TM:

$M'$ = "On input $x$:

1. If $x = 01$ then *accept*.
2. If $x \neq 10$ then *reject*.
3. If $x = 10$ simulate $M$ on $w$. If $M$ accepts $w$ then *accept*; if $M$ halts and rejects $w$ then *reject*."

If $\langle M, w\rangle \in A_{\mathsf{TM}}$ then $M$ accepts $w$ and $L(M') = \{01, 10\}$, so $\langle M'\rangle \in S$. Conversely, if $\langle M, w\rangle \notin A_{\mathsf{TM}}$ then $L(M') = \{01\}$, so $\langle M'\rangle \notin S$. Therefore, $\langle M, w\rangle \in A_{\mathsf{TM}} \iff \langle M'\rangle \in S$.

**5.13** Let $USELESS_{\mathsf{TM}} = \{\langle M\rangle \mid M$ is a TM with 1 or more useless states$\}$. We show $USELESS_{\mathsf{TM}}$ is undecidable by reducing $E_{\mathsf{TM}}$ to it. Let $R$ be a TM that decides $USELESS_{\mathsf{TM}}$ and construct a TM $S$ that decides $E_{\mathsf{TM}}$ as follows.

$S$ = "On input $\langle M, w\rangle$:

1. First we construct a TM $N$ that simulates $M$ on $w$ but ensuring that all of $M$'s original states are not useless. The TM $N$ has input alphabet $\{0, 1, 2\}$. On input 0, $N$ goes through each of the original states of $M$, except for $q_{\text{accept}}$ and $q_{\text{reject}}$, until all nonhalting states of $M$ have been used, and then enters state $q_{\text{accept}}$. On input 1, $N$ enters state $q_{\text{reject}}$. On input 2, $N$ simulates $M$ on $w$. Finally $N$ restores the original symbol to where the new symbol had been written, and branches to $M$'s start state to simulate $M$. If that simulation is about to enter $M$'s accept state, it instead enters a new state $r$.
2. Run $USELESS_{\mathsf{TM}}$ on input $\langle N\rangle$. If it accepts, *reject*. If it rejects, *accept*."

The only possible useless state of $N$ is the state $r$, because on inputs 0 and 1 it uses all other states. The only way for $N$ to use state $r$, is if $M$ accepts $w$. So, testing whether $N$ has any useless states, in effect tests whether $M$ accepts $w$. Note that the hardest part of this problem is making sure that the machine $N$ has no "unexpected" useless states.

**5.14** $L_{\mathsf{TM}} = \{\langle M, w\rangle |\ M$ on $w$ tries moving its head left when on the leftmost tape cell$\}$. This language is undecidable. To prove it, we show that if it was decidable by some machine say $R$ then we could decide $A_{\mathsf{TM}}$ with the following machine $S$:

$S$ = "On input $\langle M, w\rangle$:

1. Convert $M$ to $M'$, where $M'$ first moves its input over one square to the right and puts a special delimiter on the leftmost tape cell. Then $M'$ simulates $M$ on the input. If $M'$ ever sees a delimiter ($M$ must have tried to move left when on the leftmost tape cell), $M'$ automatically moves right and stays in the same state. If $M$ accepts, it moves its head all the way to the left and then moves left off the leftmost tape cell.
2. Run $R$, the decider for $L_{\mathsf{TM}}$, on $\langle M', w\rangle$.
3. If $R$ accepts then *accept*. If it rejects, *reject*."

TM $S$ decides $A_{\mathsf{TM}}$ since the only way $M'$ moves left from the leftmost tape cell is when $M$ accepts $w$. $S$ never attempts the move during the course of the simulation because we put the special delimiter to "intercept" such moves if made by $M$.

**5.15** Let $LM_{\mathsf{TM}} = \{\langle M, w\rangle |\ M$ ever moves left while computing on $w\}$. $LM_{\mathsf{TM}}$ is decidable. Let $M_{\mathsf{LEFT}}$ be the TM which on input $\langle M, w\rangle$ determines the number of states $n_M = |Q_M|$ of $M$ and then simulates $M$ on $w$ for $|w|+n_M+1$ steps. If $M_{\mathsf{LEFT}}$ discovers that $M$ moves left during that simulation, $M_{\mathsf{LEFT}}$ accepts $\langle M, w\rangle$. Otherwise $M_{\mathsf{LEFT}}$ rejects $\langle M, w\rangle$.

The reason that $M_{\mathsf{LEFT}}$ can reject without simulating $M$ further is as follows. Suppose $M$ does make a left move on input $w$. Let $p = q_0, q_1, \ldots, q_s$ be the shortest computation path of $M$ on $w$ ending in a left move. Because $M$ has been scanning only blanks (it's been moving right) since state $q_{|w|}$, we may remove any cycles that appear after this state and be left with a legal computation path of the machine ending in a left move. Hence $p$ has no cycles and must have length at most $|w| + n_M + 1$. Hence $M_{\mathsf{LEFT}}$ would have accepted $\langle M, w\rangle$, as desired.

**5.16** Let $B = \{\langle M\rangle |\ M$ is a two tape TM which writes a non-blank symbol on its second tape when it is simulated on some particular input $w\}$. We show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} B$ by mapping $\langle M, w\rangle$ to $\langle M'\rangle$ where $M'$ has the following description:

$M'$ = "On input $x$:

1. Simulate $M$ on $w$ only using the first tape.
2. If $M$ accepts $w$ then write `blah blah !` on the second tape."

Observe that $\langle M, w\rangle \in A_{\mathsf{TM}} \Longleftrightarrow M' \in B$.

**5.17** The *PCP* over a unary alphabet is decidable. We describe a TM $M$ that decides unary *PCP*. Given a unary *PCP* instance

$$\left\{\left[\frac{1^{a_1}}{1^{b_1}}\right], \ldots, \left[\frac{1^{a_n}}{1^{b_n}}\right]\right\}$$

$M$ = "On input $\langle a_1, b_1, \ldots, a_n, b_n\rangle$:

1. Check if $a_i = b_i$ for some $i$. If so, *accept*.
2. Check if there exist $i, j$ such that $a_i > b_i$ and $a_j < b_j$. If so, *accept*. Otherwise, *reject*."

In the first stage, $M$ first checks for a single domino which forms a solution. In the second stage, $M$ looks for two dominos which form a solution. If it finds such a pair, it can construct a solution by picking $(b_j - a_j)$ pieces of the $i^{th}$ domino, putting them together with $(a_i - b_i)$ pieces of the $j^{th}$ domino. This construction has $a_i(b_j - a_j) + a_j(a_i - b_i) = a_i b_j - a_j b_i$ 1s on the top, and $b_i(b_j - a_j) + b_j(a_i - b_i) = a_i b_j - a_j b_i$ 1s on the bottom. If neither stages of $M$ accept, the problem instance contains dominos with all the upper parts having more (or less) 1s than the lower parts. In such a case no solution can exist. Therefore $M$ rejects.

**5.18** We construct a computable function that takes a *PCP* instance of any finite alphabet and produces a binary alphabet *PCP* (*BPCP*) instance. As a result we show that *BPCP* is undecidable.

For any *PCP* instance $A$ over a finite alphabet $\{a_1, \ldots, a_n\}$, the function encodes each character $a_i$ into binary as $10^i$. The function is a mapping reduction from the general *PCP* problem to the *BPCP* problem. For any $P$ in *PCP*, the converted instance $P'$ is in *BPCP* because a match for $P$ becomes a match for $P'$ using the same set of dominos after conversion. Conversely, if $P'$ is in *BPCP*, $P$ is in *PCP*, because we can uniquely decode a match in $P'$ to a match in $P$. Therefore, the function is a mapping reduction from *PCP* to *BPCP*.

**5.19** We reduce *PCP* to $AMBIG_{\mathsf{CFG}}$, thereby proving that $AMBIG_{\mathsf{CFG}}$ is undecidable. We use the construction given in the hint in the text.

**Note:** The hint that appears in the first printing is incorrect. The correct version (which appears in the second and subsequent printings) is: Given an instance

$$P = \left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\right\},$$

$$\begin{aligned}
S &\to T \mid B \\
T &\to t_1 T \mathbf{a}_1 \mid \cdots \mid t_k T \mathbf{a}_k \mid t_1 \mathbf{a}_1 \mid \cdots \mid t_k \mathbf{a}_k \\
B &\to b_1 B \mathbf{a}_1 \mid \cdots \mid b_k B \mathbf{a}_k \mid b_1 \mathbf{a}_1 \mid \cdots \mid b_k \mathbf{a}_k ,
\end{aligned}$$

where $\mathbf{a}_1, \ldots, \mathbf{a}_k$ are new terminal symbols. We prove that this reduction works.

We show that $P$ has a solution iff the CFG is ambiguous. If $P$ has a match $t_{i_1}t_{i_2}\ldots t_{i_l} = b_{i_1}b_{i_2}\ldots b_{i_l}$, the string

$$t_{i_1}t_{i_2}\ldots t_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1} = b_{i_1}b_{i_2}\ldots b_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$$

has two different leftmost derivations, one from $T$ and one from $B$. Hence the CFG is ambiguous.

Conversely, if the CFG is ambiguous, some string $w$ has multiple leftmost derivations. All generated strings have the form $w = w_{start}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$ where $w_{start}$ contains only symbols from $P$'s alphabet. Notice that once we choose the first step in the derivation of $w$ (either $S \to T$ or $S \to B$), the following steps in the derivation are uniquely determined by the sequence $\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$. Therefore $w$ has at most two leftmost derivations:

i) $S \to T \to t_{i_1}T\mathtt{a}_{i_1} \to t_{i_1}t_{i_2}T\mathtt{a}_{i_2}\mathtt{a}_{i_1} \to \cdots \to t_{i_1}t_{i_2}\ldots t_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$, and

ii) $S \to B \to b_{i_1}T\mathtt{a}_{i_1} \to b_{i_1}b_{i_2}T\mathtt{a}_{i_2}\mathtt{a}_{i_1} \to \cdots \to b_{i_1}b_{i_2}\ldots b_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$.

If $w$ is ambiguous, $t_{i_1}t_{i_2}\ldots t_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1} = b_{i_1}b_{i_2}\ldots b_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$, and therefore $t_{i_1}t_{i_2}\ldots t_{i_l} = b_{i_1}b_{i_2}\ldots b_{i_l}$. Thus, the initial *PCP* instance, $P$, has a match.

**5.20** a. A 2DFA $M$ with $s$ states computing on an input $x$ of size $n$ has at most $s(n+2)^2$ possible configurations. Thus, on input $x$, $M$ either halts in $s(n+2)^2$ steps or loops forever. To decide $A_{\mathsf{2DFA}}$, it is enough to simulate $M$ on $x$ for $s(n+2)^2$ steps and accept if $M$ has halted and accepted $x$ during this finite duration.

b. Suppose a TM $D$ decides $E_{\mathsf{2DFA}}$. We construct a TM $E$ that decides $E_{\mathsf{TM}}$. $E$ runs as follows:

$E =$ "On input $\langle M\rangle$,

1. Construct a 2DFA M' that recognizes the language of accepting computation histories on $M$, namely, $\{\langle c_1\#c_2\#\ldots\#c_k\rangle \mid c_i$ is a configuration of $M$, $c_1$ is a start configuration, $c_k$ is an accepting configuration, $c_{i+1}$ legally follows $c_i$ for each $i\}$. A 2DFA can check whether $c_1$ is a start configuration of $M$ by verifying that it begins with a start state $q_0$ and contains legal symbols from $M$'s input alphabet. Similarly, it can check whether $c_k$ is an accepting configuration by verifying that it contains $q_{\text{accept}}$ and symbols from $M$'s tape alphabet. These two steps use only one head. To check whether $c_{i+1}$ legally follows $c_i$, the 2DFA can keep its two heads on $c_i$ and $c_{i+1}$, and compare them symbol by symbol. Notice that $L(M') = \emptyset$ iff $L(M) = \emptyset$.
2. Run $D$ on $\langle M'\rangle$. If $D$ accepts, *accept*. If $D$ rejects, *reject*."

Since $E_{\mathsf{TM}}$ is undecidable, $E_{\mathsf{2DFA}}$ is decidable.

**5.21** We formulate the problem of testing 2DIM-DFA equivalence as a language: $EQ_{\text{2DIM-DFA}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ is equivalent to } M_2\}$.
We show that if $EQ_{\text{2DIM-DFA}}$ is decidable, we can decide $A_{\text{TM}}$ by reducing $A_{\text{TM}}$ to $EQ_{\text{2DIM-DFA}}$. Therefore $EQ_{\text{2DIM-DFA}}$ is undecidable. Assume $M$ is the TM deciding $EQ_{\text{2DIM-DFA}}$. We construct $S$ that decides $A_{\text{TM}}$ as follows:

$S$ = "On input $\langle A, w\rangle$,

1. Construct a 2DIM-DFA $T$ that accepts only the accepting computation history of $A$ on $w$. The input that $T$ accepts should represent the computation history of $A$ in the form of an $m \times n$ rectangle, where $(m-2)$ is the length of the longest configuration that $A$ ever has during the computation and $(n-2)$ is the number of steps it takes to get to the accepting state. Each row of the inner cells contains a proper configuration of $A$ during its computation. In order to check if the input is a computation history of $A$ on $w$, $T$ follows a procedure similar to that given the proof of Theorem 5.9. It first checks that the first inner row contains the configuration with input $w$ and the start state at the left end. Then it checks whether the TM reaches the accept state at the last inner row. After that, it compares two consecutive rows of input to see if the two configurations follow the transition function of $A$. If the input passes all the three tests, the 2DIM-DFA accepts. Otherwise, it rejects.
2. Construct another 2DIM-DFA $E$ which always rejects. Run $M$ on $\langle T, E\rangle$ to see if $T$ and $E$ are equivalent. If $M$ accepts, *reject* because there is no computation history for $A$ on $w$. Otherwise, *accept*."

The TM $S$ decides $A_{\text{TM}}$. However, we know that $A_{\text{TM}}$ is undecidable. Therefore $EQ_{\text{2DIM-DFA}}$ is also undecidable.

**5.22** Suppose $P$ is a decidable language satisfying the properties and let $T_P$ be a TM that decides $P$. Without loss of generality assume $\langle T_\emptyset\rangle \notin P$ with $L(T_\emptyset) = \emptyset$ since we can consider $\overline{P}$ if $\langle T_\emptyset\rangle$ is in $P$. Since $P$ is not trivial, there exists a TM $M_1$ with $\langle M_1\rangle \in P$. We can decide $A_{\text{TM}}$ by constructing the following TM:

"On input $\langle M, w\rangle$:

1. Construct a TM $M_w$ that accepts on input $x$ iff $M$ accepts $w$ and $M_1$ accepts $x$.
2. Run $T_P$ on $\langle M_w\rangle$. If it accepts, *reject*. Otherwise, *accept*."

If $w \in L(M)$, $L(M_w) = L(M_1)$ and $M_w$ should be accepted by $T_P$ since $\langle M_1\rangle \in P$. If $w \notin L(M)$, $L(M_w) = \emptyset = L(T_\emptyset)$ and should be rejected by the assumption that $\langle T_\emptyset\rangle \notin P$. Therefore we have $w \in L(M)$ iff $\langle M_1\rangle \in P$. However, since $A_{\text{TM}}$ is undecidable, $P$ is not decidable either.

**5.23** First, let $P$ be the language $\{\langle M\rangle \mid M$ is a TM with 5 states$\}$. $P$ is non-trivial, and so it satisfies the second condition of Rice's Theorem but $P$ can be easily decided by checking the number of states of the input Turing Machine. Second, let $P$ be the empty set. Then it does not contain any TM and so it satisfies the first condition of Rice's Theorem, but and $P$ can be decided by a TM that always rejects.

Therefore both properties are necessary for proving $P$ undecidable.

# Chapter 6

**6.1** The following program is in LISP:

```
((LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))
(QUOTE (LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))))
```

**6.8** We build up two incomparable sets $A$ and $B$ to kill off all possible Turing-reductions, $R_1, R_2, \ldots$ . The sets start off undefined.

First, extend $A$ to be longer and longer initial portions of the empty set and $B$ to be longer and longer initial portions of ATM. Eventually, $R_1$ must get the wrong answer when trying to reduce $B$ to $A$. Then extend $A$ and $B$ interchanging the empty set and ATM until $R_1$ errs in reducing $A$ to $B$. Continue with $R_2$, etc.

# Chapter 7

**7.1**
- **a.** True
- **b.** False
- **c.** False
- **d.** True
- **e.** True
- **f.** True

**7.2**
- **a.** False
- **b.** True
- **c.** True
- **d.** True
- **e.** False
- **f.** False

**7.3** We can use the Euclidean algorithm to find the greatest common divisor.

$$\begin{aligned} 10505 &= 1274 \times 8 + 313 \\ 1274 &= 313 \times 4 + 22 \\ 313 &= 22 \times 14 + 5 \\ 22 &= 5 \times 4 + 2 \\ 5 &= 2 \times 2 + 1 \\ 2 &= 1 \times 2 + 0 \end{aligned}$$

The greatest common divisor of 10505 and 1274 is 1. Therefore they are relatively prime.

**b.**

$$\begin{aligned} 8029 &= 7289 \times 1 + 740 \\ 7289 &= 740 \times 9 + 629 \\ 740 &= 629 \times 1 + 111 \\ 629 &= 111 \times 5 + 74 \\ 111 &= 74 \times 1 + 37 \\ 74 &= 37 \times 2 + 0 \end{aligned}$$

The greatest common divisor of 8029 and 7289 is 37. Therefore they are not relatively prime.

**7.4** The table constructed using the algorithm from Theorem 7.14 is as follows:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | $T$ | $T, R$ | $S$ | $S, R, T$ |
| 2 | | $R$ | $S$ | $S$ |
| 3 | | | $T$ | $T, R$ |
| 4 | | | | $R$ |

Because $table(1, 4)$ contains $S$, the TM accepts $w$.

**7.5** The formula is not satisfiable. For any assignment of the boolean values for $x$ and $y$, it always makes one of the four clauses false. Therefore, the formula, which is a conjunction of the four clauses, is always false for any assignment of $x$ and $y$.

**7.6** P is closed under union. For any two P-languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the TMs that decide them in polynomial time. We construct a TM $M'$ that decides the union of $L_1$ and $L_2$ in polynomial time:

$M'$ = "On input $\langle w \rangle$:

1. Run $M_1$ on $w$. If it accepts, *accept*.
2. Run $M_2$ on $w$. If it accepts, *accept*. Otherwise, *reject*."

$M'$ accepts $w$ if and only if either $M_1$ and $M_2$ accept $w$. Therefore, $M'$ decides the union of $L_1$ and $L_2$. Since both stages take polynomial time, the algorithm runs in polynomial time.

P is closed under concatenation. For any two P-languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the TMsthat decide them in polynomial time. We construct a TM $M'$ that decides the concatenation of $L_1$ and $L_2$ in polynomial time:

$M'$ = "On input $\langle w \rangle$:

1. For each way to cut $w$ into two substrings $w = w_1 w_2$:
2. Run $M_1$ on $w_1$ and $M_2$ on $w_2$. If both accept, *accept*.
3. If $w$ is not accepted after trying all the possible cuts, *reject*."

$M'$ accepts $w$ if and only if $w$ can be written as $w_1 w_2$ such that $M_1$ accepts $w_1$ and $M_2$ accepts $w_2$. Therefore, $M'$ decides the concatenation of $L_1$ and $L_2$. Since stage 2 runs in polynomial time and is repeated at most $O(n)$ times, the algorithm runs in polynomial time.

P is closed under complement. For any P-language $L$, let $M$ be the TM that decides it in polynomial time. We construct a TM $M'$ that decides the complement of $L$ in polynomial time:

$M'$ = "On input $\langle w \rangle$:

1. Run $M$ on $w$.
2. If $M$ accepts, *reject*. If it rejects, *accept*."

$M'$ decides the complement of $L$. Since $M$ runs in polynomial time, $M'$ also runs in polynomial time.

**7.7** NP is closed under union. For any two NP-languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the NTMsthat decide them in polynomial time. We construct a NTM $M'$ that decides the union of $L_1$ and $L_2$ in polynomial time:

$M'$ = "On input $\langle w \rangle$:

1. Run $M_1$ on $w$. If it accepts, *accept*.
2. Run $M_2$ on $w$. If it accepts, *accept*. Otherwise, *reject*."

In both stages 1 and 2, $M'$ uses its nondeterminism when the machines being run make nondeterministic steps. $M'$ accepts $w$ if and only if either $M_1$ and $M_2$ accept $w$. Therefore, $M'$ decides the union of $L_1$ and $L_2$. Since both stages take polynomial time, the algorithm runs in polynomial time.

NP is closed under concatenation. For any two NP-languages $L_1$ and $L_2$, let $M_1$ and $M_2$ be the NTMsthat decide them in polynomial time. We construct a NTM $M'$ that decides the concatenation of $L_1$ and $L_2$ in polynomial time:

$M'$ = "On input $\langle w \rangle$:

1. For each way to cut $w$ into two substrings $w = w_1 w_2$:
2. Run $M_1$ on $w_1$.
3. Run $M_2$ on $w_2$. If both accept, *accept*; otherwise continue with the next choice of $w_1$ and $w_2$.
4. If $w$ is not accepted after trying all the possible cuts, *reject*."

In both stages 2 and 3, $M'$ uses its nondeterminism when the machines being run make nondeterministic steps. $M'$ accepts $w$ if and only if $w$ can be expressed as $w_1 w_2$ such that $M_1$ accepts $w_1$ and $M_2$ accepts $w_2$. Therefore, $M'$ decides the concatenation of $L_1$ and $L_2$. Since stage 2 runs in polynomial time and is repeated for at most $O(n)$ time, the algorithm runs in polynomial time.

**7.8** If we use unary encoding, the input number $n$ has length $n$. An algorithm can determine the primality of $n$ by checking sequentially whether $n$ is divisible by any integer from 2 to $n$. If any of the numbers divide $n$, *reject*. Otherwise *accept*. The time to do the division is polynomial in $n$, and the total number of divisions done is at most $n$. Hence the algorithm runs in polynomial (in $n$) time. Therefore, *UNARY-PRIMES* is in P.

**7.9** The algorithm given in page 145 runs in $O(n^3)$ time. Stage 1 takes at most $O(n)$ steps to locate and mark the start node. Stage 2 causes at most $n+1$ repetitions, because each repetition except the last marks at least one additional node. Each execution of stage 3 uses at most $O(n^3)$ steps because $G$ contains at most $n$ to be checked and for each checked node, examining all adjacent nodes to see whether any have been marked uses at most $O(n^2)$ steps. Therefore in total, stages 2 and 3 take $O(n^4)$ time. Stage 4 uses $O(n)$ steps to scan all nodes. Therefore, the algorithm runs in $O(n^4)$ time and *CONNECTED* is in P.

**7.10** We construct a TM $M$ that decides *TRIANGLE* in polynomial time.

$M$ = "On input $\langle G \rangle$ where $G$ is a graph:

1. For each triple of vertices $v_1, v_2, v_3$ in $G$:
2. If edges $(v_1, v_2)$, $(v_1, v_3)$, and $(v_2, v_3)$, are all edges of $G$, *accept*.
3. No triangle has been found in $G$, so *reject*."

A graph with $m$ vertices has $\binom{m}{3} = \frac{m!}{3!(m-3)!} = O(m^3)$ triples of vertices. Therefore, stage 2 will be repeated at most $O(m^3)$ times. In addition, each stage can be implemented to run in polynomial time. Therefore, *TRIANGLE* $\in$ P.

**7.11** A nondeterministic polynomial time algorithm for *ISO* operates as follows:

"On input $\langle G, H \rangle$ where $G$ and $H$ are undirected graphs:

1. Let $m$ be the number of nodes of $G$ and $H$. If they don't have the same number of nodes, *reject*.
2. Nondeterministically select a permutation $\pi$ of $m$ elements.
3. For each pair of nodes $x$ and $y$ of $G$ check that $(x, y)$ is an edge of $G$ iff $(\pi(x), \pi(y))$ is an edge of $H$. If all agree, *accept*. If any differ, *reject*."

Stage 2 can be implemented in polynomial time nondeterministically. Stage 3 takes polynomial time. Therefore $ISO \in$ NP.

**7.12** The most obvious algorithm multiplies $a$ by itself $b$ times then compares the result to $c$, modulo $p$. That uses $b-1$ multiplications which is exponential in the length of $b$. Hence this algorithm doesn't run in polynomial time. Here is one that does:

"On input $\langle a, b, c, p\rangle$, four binary integers:

1. Let $r \leftarrow 1$.
2. Let $b_1, \ldots, b_k$ be the bits in the binary representation of $b$.
3. For $i = 1, \ldots, k$:
4. If $b_i = 0$, let $r \leftarrow r^2 \mod p$.
5. If $b_i = 1$, let $r \leftarrow ar^2 \mod p$.
6. If $(c \mod p) = (r \mod p)$, *accept*; otherwise *reject*."

The algorithm is called *repeated squaring*. Each of its multiplications can be done in polynomial time in the standard way, because the numbers are no larger than $p$. The total number of multiplications is proportional to the number of bits in $b$. Hence the total running time is polynomial.

**7.13** We show that P is closed under the star operation by dynamic programming. Let $A \in$ P, and $M$ be the TM deciding $A$ in polynomial time. On input $w = w_1 \cdots w_n$, use the following procedure $ST$ to construct a table $T$ such that $T[i,j] = 1$ if $w_i \cdots w_j \in A^*$ and $T[i,j] = 0$ otherwise, for all $1 \le i \le j \le n$.

$ST$ = "On input $w = w_1 \cdots w_n$:

1. If $w = \varepsilon$, *accept*. [handle $w = \varepsilon$ case]
2. Initialize $T[i,j] = 0$ for $1 \le i \le j \le n$.
3. For $i = 1$ to $n$, [test each substring of length 1]
4. Set $T[i,i] = 1$ if $w_i \in A$.
5. For $l = 2$ to $n$, [$l$ is the length of the substring]
6. For $i = 1$ to $n - l + 1$, [$i$ is the substring start position]
7. Let $j = i + l - 1$, [$j$ is the substring end position]
8. If $w_i \cdots w_j \in A$, set $T[i,j] = 1$.
9. For $k = i$ to $j - 1$, [$k$ is the split position]
10. If $T[i,k] = 1$ and $T[k,j] = 1$, set $T[i,j] = 1$.
11. *Accept* if $T[1,n] = 1$; otherwise *reject*."

Each stage of the algorithm takes polynomial time, and $ST$ runs for $O(n^3)$ stages, so the algorithm runs in polynomial time.

**7.14** Let NTM $M$ decide $A$ in nondeterministic polynomial time. The following NTM $N$ decides $A^*$.

$N$ = "On input $w$:

1. If $w = \varepsilon$, *accept*.
2. Nondeterministically choose $k$, where $1 \leq k \leq n$.
3. Nondeterministically select strings $w_1, w_2, \ldots, w_k$, each of length at most $n$.
4. If $w \neq w_1 \cdots w_k$, then *reject*(on this branch).
5. Simulate (using nondeterminism) $M$ on each $w_i$.
6. If $M$ accepts each $w_i$, then *accept*; otherwise *reject*."

Each stage takes polynomial time, so the algorithm runs in polynomial time.

**7.15** The reduction fails because it would result in an exponential unary instance, and therefore would not be a polynomial time reduction.

We give a polynomial time algorithm for this problem using dynamic programming. The input to the algorithm is $\langle S, t\rangle$ where $S$ is a set of numbers $\{x_1, \ldots, x_k\}$. The algorithm operates by constructing a list $L_i$ of numbers for $0 \leq i \leq k$ giving the set of possible values that a subset of the numbers $\{x_1, \ldots, x_i\}$ could sum to. Initially $L_0 = \{0\}$. For each $i$ the algorithm constructs $L_i$ from $L_{i-1}$ by assigning $L_i = L_{i-1} \cup \{a + x_i \mid a \in L_{i-1}\}$. Finally, the algorithm accepts if $t \in L_k$, and rejects otherwise.

**7.16** **a.** Follow the marking algorithm for recognizing *PATH* while additionally keeping track of the length of the shortest paths discovered. Here is a more detailed description:

"On input $\langle G, a, b, k\rangle$ where $G$ is a directed graph on $m$ nodes where $G$ has nodes $a$ and $b$:

1. Place a mark with label 0 on node $a$.
2. For each $i$ from 0 to $m$:
3. Scan all the edges of $G$. If an edge $(s, t)$ is found going from a node $s$ marked with $i$ to an unmarked node $b$, mark node $t$ with $i + 1$.
4. If $t$ is marked with a value of at most $k$, *accept*. Otherwise, *reject*."

**b.** First, *LPATH* $\in$ NP because a nondeterministic machine can guess and verify a simple path of length at least $k$ from $a$ to $b$. Next, *UHAMPATH* $\leq_{\mathrm{P}}$ *LPATH*, because the following TM $F$ computes the reduction $f$.

$F$ = "On input $\langle G, a, b\rangle$, where $G$ is an undirected graph, and $a$ and $b$ are nodes of $G$.

1. Let $k$ be the number of nodes of $G$.
2. Output $\langle G, a, b, k\rangle$."

If $\langle G, a, b\rangle \in$ *UHAMPATH*, G contains a Hamiltonian path of length $k$ from $a$ to $b$, thus $\langle G, a, b, k\rangle \in$ *LPATH*. If $\langle G, a, b, k\rangle \in$ *LPATH*, G contains a simple path of length $k$ from $a$ to $b$. Since $G$ has $k$ nodes, the path is Hamiltonian. Thus $\langle G, a, b\rangle \in$ *UHAMPATH*.

**7.17** Let $A$ be any language in P except $A = \emptyset$ and $A = \Sigma^*$. To show that $A$ is NP-complete, we show that $A \in$ NP and that every $B \in$ NP is polynomial time reducible to $A$. The first condition holds because $A \in$ P so $A \in$ NP. To demonstrate that the second condition is true, let $x_{\text{in}} \in A$ and $x_{\text{out}} \notin A$ be two strings that are guaranteed to exist by virtue of our assumptions about $A$. The assumption that P = NP implies that $B$ is recognized by a polynomial time TM $M$. A polynomial time reduction from $B$ to $A$ simulates $M$ to determine whether its input $w$ is a member of $B$, then outputs $x_{\text{in}}$ or $x_{\text{out}}$ accordingly.

**7.19** On input $\phi$, a nondeterministic polynomial time machine can guess two assignments and accept if both assignments satisfy $\phi$. Thus *DOUBLE-SAT* is in NP. We show *SAT* $\leq_{\text{P}}$ *DOUBLE-SAT*. The following TM $F$ computes the polynomial time reduction $f$.

$F$ = "On input $\langle\phi\rangle$, a Boolean formula with variables $x_1, x_2, \ldots, x_m$.

1. Let $\phi'$ be $\phi \wedge (x \vee \overline{x})$, where $x$ is a new variable.
2. Output $\langle\phi'\rangle$"

If $\phi \in$ *SAT*, $\phi'$ has at least two satisfying assignments because we can obtain two assignments from the original assignment of $\phi$ by changing the value of $x$. If $\phi' \in$ *DOUBLE-SAT*, $\phi$ is also satisfiable, because $x$ does not appear in $\phi$. Therefore $\phi \in$ *SAT* if and only if $f(\phi) \in$ *DOUBLE-SAT*.

**7.20** First, note that we can compose two permutations in polynomial time. Composing $q$ with itself $t$ times requires $t$ compositions if done directly, thereby giving a running time that is exponential in the length of $t$ when $t$ is represented in binary. We can fix this problem using a common technique for fast exponentiation. Calculate $q^1, q^2, q^4, q^8, \ldots, q^k$, where $k$ is the largest power of 2 that is smaller than $t$. Each term is the square of the previous term. To find $q^t$ we compose the previously computed permutations corresponding to the 1s in $t$s binary representation. The total number of compositions is now at most the length of $t$, so the algorithm runs in polynomial time.

**7.21** *HALF-CLIQUE* $\in$ NP because a nondeterministic machine can guess and verify a clique with at least $m/2$ nodes in polynomial time. We show that *CLIQUE* $\leq_{\text{P}}$ *HALF-CLIQUE*. The following TM $F$ computes the reduction $f$.

$F$ = "On input $\langle G, k\rangle$ where $G$ is an undirected graph with $m$ nodes and $k$ is an integer.

1. If $k = m/2$ output $\langle G\rangle$.
2. If $k < m/2$, construct $G'$ by adding a complete graph with $m - 2k$ nodes and connecting them to all nodes in $G$. Output $\langle G'\rangle$.

3. If $k > m/2$, construct $G''$ by adding $2k - m$ isolated nodes. Output $\langle G'' \rangle$."

When $k < m/2$, if $G$ has a $k$-clique, $G'$ has a clique of size $k + (m - 2k) = (2m - 2k)/2$, thus $G' \in$ *HALF-CLIQUE*. If $G'$ has a $k + (m - 2k)$-clique, then at most $m - 2k$ of them come from the new nodes, thus $G$ must contain a $k$-clique. When $k > m/2$, if $G$ has a $k$-clique, $G''$ has a clique of size $(m + 2k - m)/2 = k$, thus $G'' \in$ *HALF-CLIQUE*. If $G'' \in$ *HALF-CLIQUE*, $G$ must contain a $k$-clique because the half-clique cannot include any of the new isolated nodes.

**7.22** a. In an $\neq$-assignment each clause has at least one literal assigned 1 and at least one literal assigned 0. The negation of an $\neq$-assignment preserves this property, so it too is an $\neq$-assignment.

b. To prove that the given reduction works, we need to show that if the formula $\phi$ is mapped to $\phi'$, then $\phi$ is satisfiable (in the ordinary sense) iff $\phi'$ has an $\neq$-assignment. First, if $\phi$ is satisfiable, we can obtain an $\neq$-assignment for $\phi'$ by extending the assignment to $\phi$ so that we assign $z_i$ to 1 if both literals $y_1$ and $y_2$ in clause $c_i$ of $\phi$ are assigned 0. Otherwise we assign $z_i$ to 0. Finally, we assign $b$ to 0. Second, if $\phi'$ has an $\neq$-assignment we can find a satisfying assignment to $\phi$ as follows. By part (a) we may assume the $\neq$-assignment assigns $b$ to 0 (otherwise, negate the assignment). This assignment cannot assign all of $y_1$, $y_2$, and $y_3$ to 0, because doing so would force one of the clauses in $\phi'$ to have all 0s. Hence, restricting this assignment to the variables of $\phi$ gives a satisfying assignment.

c. Clearly, $\neq$*SAT* is in NP. Hence, it is NP-complete because *3SAT* reduces to it.

**7.23** First, *MAX-CUT* $\in$ NP because a nondeterministic algorithm can guess the cut and check that it has size at least $k$ in polynomial time.

Second, we show $\neq$ *3SAT* $\leq_{\mathrm{P}}$ *MAX-CUT*. Use the reduction suggested in the hint. For clarity, though, use $c$ instead of $k$ for the number of clauses. Hence, given an instance $\phi$ of $\neq$ *3SAT* with $v$ variables and $c$ clauses. We build a graph $G$ with $6cv$ nodes. Each variable $x_i$ corresponds to $6c$ nodes; $3c$ labeled $x_i$ and $3c$ labeled $\overline{x_i}$. Connect each $x_i$ node with each $\overline{x_i}$ node for a total of $9c^2$ edges. For each clause in $\phi$ select three nodes labeled by the literal in that clause and connect them by edges. Avoid selecting the same node more than once (each label has $3c$ copies, so we cannot run out of nodes). Let $k$ be $9c^2v + 2c$. Output $\langle G, k \rangle$. Next we show this reduction works.

We show that $\phi$ has a $\neq$-assignment iff $G$ has a cut of size at least $k$. Take an $\neq$-assignment for $\phi$. It yields a cut of size $k$ by placing all nodes corresponding to true variables on one side and all other nodes on the other side. Then the cut contains all $9c^2v$ edges between literals and their negations and two edges

for each clause, because it contains at least one true and one false literal, yielding a total of $9c^2v + 2c = k$ edges.

For the converse, take a cut of size $k$. Observe that a cut can contain at most two edges for each clause, because at least two of the corresponding literals must be on the same side. Thus, the clause edges can contribute at most $2c$ to the cut. The graph $G$ has only $9c^2v$ other edges, so if $G$ has a cut of size $k$, all of those other edges appear in it and each clause contributes its maximum. Hence, all nodes labeled by the same literal occur on the same side of the cut. By selecting either side of the cut and assigning its literals true, we obtain an assignment. Because each clause contributes two edges to the cut, it must have nodes appearing on both sides and so it has at least one true and one false literal. Hence $\phi$ has a $\neq$-assignment.

(A minor glitch arises in the above reduction if $\phi$ has a clause that contains both $x$ and $\overline{x}$ for some variable $x$. In that case $G$ would need to be a multigraph, because it would contain two edges joining certain nodes. We can avoid this situation by observing that such clauses can be removed from $\phi$ without changing its $\neq$-satisfiability.)

**7.24** To show that all problems in NP are polynomial time reducible to $D$ we show that $3SAT \leq_{\mathrm{P}} D$. Given a 3cnf formula $\phi$ we construct a polynomial $p$ with the same variables. We represent the variable $x$ in $\phi$ by $x$ in $p$ and its negation $\overline{x}$ by $1 - x$. Next we represent the Boolean operations $\wedge$ and $\vee$ in $\phi$ by arithmetic operations that simulate them, as follows. The Boolean expression $y_1 \wedge y_2$ becomes $y_1 y_2$ and $y_1 \vee y_2$ becomes $(1 - (1 - y_1)(1 - y_2))$. Hence, the clause $(y_1 \vee y_2 \vee y_3)$ becomes $(1 - (1 - y_1)(1 - y_2)(1 - y_3))$. The conjunction of the clauses of $\phi$ becomes a product of their individual representations. The result of this transformation is a polynomial, $q$, which simulates $\phi$ in the sense they both have the same value when the variables have Boolean assignments. Thus the polynomial $1 - q$ has an integral (in fact Boolean) root if $\phi$ is satisfiable. However, $1 - q$ does not meet the requirements of $p$, because $1 - q$ may have an integral (but non-Boolean) root even when $\phi$ is unsatisfiable. To prevent that situation, we observe that the polynomial $r = (x_1(1 - x_1))^2(x_2(1 - x_2))^2 \cdots (x_m(1 - x_m))^2$ is 0 only when the variables $x1, \ldots, x_l$ take on Boolean values, and is positive otherwise. Hence polynomial $p = (1 - q)^2 + r$ has an integral root iff $\phi$ is satisfiable.

**7.25** $U$ is in NP because on input $\langle d, x, 1^t \rangle$ a nondeterministic algorithm can simulate $M_d$ on $x$ (making nondeterministic branches when $M_d$ does) for $t$ steps and accept if $M_d$ accepts. Doing so takes time polynomial in the length of the input because $t$ appears in unary.

To show $3SAT \leq_{\mathrm{P}} U$, let $M$ be a NTM that decides $3SAT$ in time $cn^k$ for some constants $c$ and $k$. Given a formula $\phi$, transform it into $w = \langle \langle M \rangle, \phi, 1^{c|\phi|^k} \rangle$. By the definition of $M$, $w \in U$ if and only if $\phi \in 3SAT$.

**7.26** *PUZZLE* is in NP because we can guess a solution and verify it in polynomial time. We reduce *3SAT* to *PUZZLE* in polynomial time. Given a Boolean formula $\phi$, convert it to a set of cards as follows.

Let $x_1, x_2, \ldots, x_m$ be the variables of $\phi$ and let $c_1, c_2, \ldots, c_l$ be the clauses. Say that $z \in c_j$ if $z$ is one of the literals in $c_j$. Let each card have column 1 and column 2 with $l$ possible holes in each column numbered 1 through $l$. When placing a card in the box, say that a card is *face up* if column 1 is on the left and column 2 is on the right; otherwise *face down*. The following algorithm $F$ computes the reduction.

$F$ = "On input $\phi$:

1. Create one card for each $x_i$ as follows: in column 1 punch out all holes except any hole $j$ such that $x_i \in c_j$; in column 2 punch out all holes except any hole $j'$ such that $\overline{x_i} \in c_{j'}$.
2. Create an *extra* card with all holes in column 1 punched out and no hole in column 2 punched out.
3. Output the description of the cards created."

Given a satisfying assignment for $\phi$, a solution to the *PUZZLE* can be constructed as follows. For each $x_i$ assigned True (False), put its card face up (down), and put the extra card face up. Then, every hole in column 1 is covered because the associated clause has a literal assigned True, and every hole in column 2 is covered by the extra card.

Given a solution to the *PUZZLE*, a satisfying assignment for $\phi$ can be constructed as follows. Flip the deck so that the extra card covers column 2. Assign the variables according to the correponding cards, True for up and False for down. Because every hole in column 1 is covered, every clause must contain at least one literal assigned True in this assignment. Hence it satisfies $\phi$.

**7.27** *SET-SPLITTING* in in NP because a coloring of the elements can be guesses and verified to have the desired properties in polynomial time. We give a polynomial time reduction $f$ from *3SAT* to *SET-SPLITTING* as follows.

Given a 3cnf $\phi$, we set $S = \{x_1, \overline{x_1}, \ldots, x_m, \overline{x_m}, y\}$. In other words, $S$ contains an element for each literal (two for each variable) and a special element $y$. For every clause $c_i$ in $\phi$, let $C_i$ be a subset of $S$ containing the elements corresponding to the literals in $c_i$ and the special element $y \in S$. We also add subsets $C_{x_i}$ for each literal, containing elements $x_i$ and $\overline{x_i}$. Then $C = \{C_1, \ldots, C_l, C_{x_1}, \ldots, C_{x_l}\}$.

If $\phi$ is satisfiable, consider a satisfying assignment. If we color all the true literals red, all the false ones blue, and $y$ blue, then every subset $C_i$ of $S$ has at least one red element (because $c_i$ of $\phi$ is "turned on" by some literal) and it also contains one blue element ($y$). In addition, every subset $C_{x_i}$ has exactly one element red and one blue, so that the system $\langle S, C\rangle \in$ *SET-SPLITTING*.

If $\langle S, C\rangle \in$ *SET-SPLITTING*, then look at the coloring for $S$. If we set the literals to true which are colored differently from $y$, and those to false which are the same color as $y$ we obtain a satisfying assignment to $\phi$.

**7.28** Let $F = \{\langle a, b, c\rangle \mid a, b, c$ are binary integers and $a = pq$ for $b \leq p \leq c\}$. We see that $F \in$ NP because a nondeterministic algorithm can guess $p$ and check that $p$ divides $a$ and that $b \leq p \leq c$ in polynomial time. If P = NP then $F \in$ P. In that case, we can locate a factor of $a$ by successively running $F$ on smaller and smaller intervals $b, c$, in effect performing a binary search on the interval 1 to $a$. The running time of that procedure is the log of the size of the interval, and thus is polynomial in the length of $a$.

**7.29** Assuming P = NP, we have a polynomial time algorithm deciding *SAT*. To produce a satisfying assignment for a satisfiable formula $\phi$, substitute $x_1 = 0$ and $x_1 = 1$ in $\phi$ and test the satisfiability of the two resulting formulas $\phi_0$ and $\phi_1$. At least one of these must be satisfiable because $\phi$ is satisfiable. At least one of these must be satisfiable because $\phi$ is satisfiable. If $\phi_0$ is satisfiable, pick $x_1 = 0$; otherwise $\phi_1$ must be satisfiable and pick $x_1 = 1$. That gives the value of $x_1$ in a satisfying assignment. Make that substitution permanent and determine a value for $x_2$ in that satisfying assignment in the same way. Continue until all variables been substituted.

**7.30** If P = NP, then *CLIQUE* is recognizable in polynomial time. We show how to compute *MAX-CLIQUE* using *CLIQUE*. Let $m$ be the number of nodes in the input graph. For each $i$ from 1 to $m$, test whether there exists a clique of size $i$ using the polynomial time algorithm for *CLIQUE*. Output the largest such $i$ for which a clique exists.

To find the maximum clique, we start with $i$, the maximum clique size. Remove one node and see if there is still a clique of size $i$. If not, restore that node and remove another. If so, iterate the process until we are left with a graph of $i$ nodes, which must be a clique. This process takes at most $m$ trials to find which node to remove, and at most $m$ nodes need to be removed. The total running time is therefore polynomial.

**7.31** Test whether a path exists from the start state to each non-accepting state. [this problem is too easy—should have been only an exercise]

**7.32** If we used $2 \times 2$ windows, we might not detect certain invalid computation histories where multiple heads occurred in the same row. For example, if the NTM had the nondeterministic choice of moving its head left or right when in a certain state reading some symbol, the next configuration must have one of these possibilities, but not both simultaneously. However, using a $2 \times 2$ window, a row that contained two state symbols corresponding to each possible next move would not appear incorrect, because each window would itself be legal.

Furthermore, once two heads appeared in a single row, they could collaborate to make the final row accepting, even though no actual accepting computation existed.

**7.33** The given algorithm does use exponential time, but a different, polynomial time algorithm could conceivably exist. Hence, *SAT* might have polynomial time complexity. The error in the proof occurs in the sentence, "Thus *SAT* has exponential time complexity" because that conclusion doesn't follow logically from the previous statements.

**7.34** *3COLOR* is in NP because a coloring can be verified in polynomial time. We show *3SAT* $\leq_{\mathrm{P}}$ *3COLOR*. Let $\phi = c_1 \wedge c_2 \wedge \cdots \wedge c_l$ be a 3cnf formula over variables $x_1, x_2, \cdots, x_m$, where the $c_i$'s are the clauses. We build a graph $G_\phi$ containing $2m + 6l + 3$ nodes: 2 nodes for each variable; 6 nodes for each clause; and 3 extra nodes. We describe $G_\phi$ in terms of the subgraph gadgets given in the Hint.

$G_\phi$ contains a variable gadget for each variable $x_i$, two OR-gadgets for each clause, and one palette gadget. The four bottom nodes of the OR-gadgets will be merged with other nodes in the graph, as follows. Label the nodes of the palette gadget T, F, and R. Label the nodes in each variable gadget + and − and connect each to the R node in the palette gadget. In each clause, connect the top of one of the OR-gadgets to the F node in the palette. Merge the bottom node of that OR-gadget with the top node of the other OR-gadget. Merge the three remaining bottom nodes of the two OR-gadgets with corresponding nodes in the variable gadgets so that if a clause contains the literal $x_i$, one of its bottom nodes is merged with the + node of $x_i$ whereas if the clause contains the literal $\overline{x_i}$, one of its bottom nodes is merged with the − node of $x_i$.

To show the construction is correct, we first demonstrate that if $\phi$ is satisfiable, the graph is 3-colorable. The three colors are called T, F, and R. Color the palette with its labels. For each variable, color the + node T and the − node F if the variable is True in a satisfying assignment; otherwise reverse the colors. Because each clause has one True literal in the assignment, we can color the nodes of the OR-gadgets of that clause so that the node connected to the F node in the palette is not colored F. Hence we have a proper 3-coloring.

If we start out with a 3-coloring, we can obtain a satisfying assignment by taking the colors assigned to the + nodes of each variable. Observe that neither node of the variable gadget can be colored R, because all variable nodes are connected to the R node in the palette. Furthermore, if both bottom nodes of an OR-gadget are colored F, the top node must be colored F, and hence, each clause contain a true literal. Otherwise, the three bottom nodes that were merged with variable nodes would be colored F, and then both top nodes would be colored F, but one of the top nodes is connected to the F node in the palette.

**7.36** We build an NFA in polynomial time that accepts all non-satisfying assignments. The NFA operates by guessing a clause is not satisfied by the assignment and then reading the input to check that the clause is indeed not

satisfied. More precisely, the NFA contains $l(m+1)+1$ states, where $l$ is the number of clauses and $m$ is the number of variables in $\phi$. Each clause $c_j$ is represented by $m+1$ states $q_{j,1}, \ldots, q_{j,m+1}$. The start state $q_0$ branches nondeterministically to each state $q_{1,j}$ on $\varepsilon$. Each state $q_{i,j}$ branches to $q_{i,j+1}$ for $j \leq m$. The label on that transition is 0 if $x_i$ appears in clause $c_j$, and is 1 if $\overline{x_i}$ appears in $c_j$. If neither literals appear in $c_j$, the label on the transition is 0,1. Each state $q_{i,m+1}$ is an accept state.

If we could minimize this NFA in polynomial time, we would be able to determine whether it accepts all strings of length $m$, and hence whether $\phi$ is satisfiable, in polynomial time. That would imply P = NP.

**7.37** The clause $(x \vee y)$ is logically equivalent to each of the expressions $(\overline{x} \rightarrow y)$ and $(\overline{y} \rightarrow x)$. We represent the 2cnf formula $\phi$ on the variables $x_1, \ldots, x_m$ by a directed graph $G$ on $2m$ nodes labeled with the literals over these variables. For each clause in $\phi$, place two edges in the graph corresponding to the the two implications above. Then we show that $\phi$ is satisfiable iff $G$ doesn't contain a cycle containing both $x_i$ and $\overline{x_i}$ for any $i$. We'll call such a cycle an ***inconsistency cycle***. Testing whether $G$ contains an inconsistency cycle is easily done in polynomial time with a marking algorithm, or a depth-first search algorithm.

We first show that if $G$ contains an inconsistency cycle, no satisfying assignment can exist. The sequences of implications in the inconsistency cycle yields the logical equivalence $x_i \leftrightarrow \overline{x_i}$ for some $i$, and that is contradictory. Thus $\phi$ is unsatisfiable.

Next, we show that if $G$ doesn't contain an inconsistency cycle, $\phi$ is satisfiable. Write $x \xrightarrow{*} y$ if $G$ contains a path from node $x$ to node $y$. Because $G$ contains the two designated edges for each clause in $\phi$, we have $x \xrightarrow{*} y$ iff $\overline{y} \xrightarrow{*} \overline{x}$. Now we construct the satisfying assignment.

Pick any variable $x_i$. We cannot have both $x_i \xrightarrow{*} \overline{x_i}$ and $\overline{x_i} \xrightarrow{*} x_i$ because $G$ doesn't contain an inconsistency cycle. Select literal $x_i$ if $\overline{x_i} \xrightarrow{*} x_i$ is false, and otherwise select $\overline{x_i}$. Assign the selected literal and all implied literals (those reachable along paths from the selected node) to be True. Note that we never assign both $x_j$ and $\overline{x_j}$ True because if $x_i \xrightarrow{*} x_j$ and $x_i \xrightarrow{*} \overline{x_j}$ then $x_j \xrightarrow{*} \overline{x_i}$ and hence $x_i \xrightarrow{*} \overline{x_i}$ thus we would not have selected literal $x_i$ (similarly for $\overline{x_i}$). Then, we remove all nodes labeled with assigned literals or their complements from $G$, and repeat this paragraph until all variables are assigned.

The resulting assignment satisfies $\phi$ because it satisfies every clause. As soon as one of the literals in a clause is assigned False, the other must be assigned True because $G$ contains an arrow from the negation of the falsified literal to that other literal.

CENGAGE Learning

# Chapter 0

**0.1** **a.** The odd positive integers.
**b.** The even integers.
**c.** The even positive integers.
**d.** The positive integers which are a multiple of 6.
**e.** The palindromes over $\{\texttt{0},\texttt{1}\}$.
**f.** The empty set.

**0.2** **a.** $\{1, 10, 100\}$.
**b.** $\{n|\ n > 5\}$.
**c.** $\{1, 2, 3, 4\}$.
**d.** $\{\texttt{aba}\}$.
**e.** $\{\varepsilon\}$.
**f.** $\emptyset$.

**0.3** **a.** No.
**b.** Yes.
**c.** $A$.
**d.** $B$.
**e.** $\{(\texttt{x},\texttt{x}), (\texttt{x},\texttt{y}), (\texttt{y},\texttt{x}), (\texttt{y},\texttt{y}), (\texttt{z},\texttt{x}), (\texttt{z},\texttt{y})\}$.
**f.** $\{\emptyset, \{\texttt{x}\}, \{\texttt{y}\}, \{\texttt{x},\texttt{y}\}\}$.

**0.4** $A \times B$ has $ab$ elements, because each element of $A$ is paired with each element of $B$, so $A \times B$ contains $b$ elements for each of the $a$ elements of $A$.

**0.5** $\mathcal{P}(C)$ contains $2^c$ elements because each element of $C$ may either be in $\mathcal{P}(C)$ or not in $\mathcal{P}(C)$, and so each element of $C$ doubles the number of subsets of $C$. Alternatively, we can view each subset $S$ of $C$ as corresponding to a binary string $b$ of length $c$, where $S$ contains the $i$th element of $C$ iff the $i$th place of $b$ is $\texttt{1}$. There are $2^c$ strings of length $c$ and hence that many subsets of $C$.

**0.6** **a.** $f(2) = 7$.
**b.** The range $= \{6, 7\}$ and the domain $= \{1, 2, 3, 4, 5\}$.
**c.** $g(2, 10) = 6$.
**d.** The range $= \{1, 2, 3, 4, 5\} \times \{6, 7, 8, 9, 10\}$ and the domain $= \{6, 7, 8, 9, 10\}$.
**e.** $f(4) = 7$ so $g(4, f(4)) = g(4, 7) = 8$.



© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**0.7** The underlying set is $\mathcal{N}$ in these examples.

**a.** Let $R$ be the "within 1" relation, that is, $R = \{(a, b) \mid |a - b| \leq 1\}$.

**b.** Let $R$ be the "less than or equal to" relation, that is, $R = \{(a, b) \mid a \leq b\}$.

**c.** Finding a $R$ that is symmetric and transitive but not reflexive is tricky because of the following "near proof" that $R$ cannot exist! Assume that $R$ is symmetric and transitive and chose any member $x$ in the underlying set. Pick any other member $y$ in the underlying set for which $(x, y) \in R$. Then $(y, x) \in R$ because $R$ is symmetric and so $(x, x) \in R$ because $R$ is transitive, hence $R$ is reflexive. This argument fails to be an actual proof because $y$ may fail to exist for $x$.

Let $R$ be the "neither side is 1" relation, $R = \{(a, b) \mid a \neq 1 \text{ and } b \neq 1\}$.

**0.10** Let $G$ be any graph with $n$ nodes where $n \geq 2$. The degree of every node in $G$ is one of the $n$ possible values from 0 to $n - 1$. We would like to use the pigeon hole principle to show that two of these values must be the same, but number of possible values is too great. However, not all of the values can occur in the same graph because a node of degree 0 cannot coexist with a node of degree $n - 1$. Hence $G$ can exhibit at most $n - 1$ degree values among its $n$ nodes, so two of the values must be the same.

**0.11** The error occurs in the last sentence. If $H$ contains at least 3 horses, $H_1$ and $H_2$ contain a horse in common, so the argument works properly. But, if $H$ contains exactly 2 horses, then $H_1$ and $H_2$ each have exactly 1 horse, but do not have a horse in common. Hence we cannot conclude that the horse in $H_1$ has the same color as the horse in $H_2$. So the 2 horses in $H$ may not be colored the same.

**0.12** **a.** *Basis:* Let $n = 0$. Then, $S(n) = 0$ by definition. Furthermore, $\frac{1}{2}n(n + 1) = 0$. So $S(n) = \frac{1}{2}n(n + 1)$ when $n = 0$.

*Induction:* Assume true for $n = k$ where $k \geq 0$ and prove true for $n = k + 1$. We can use this series of equalities:

$$\begin{aligned}
S(k+1) &= 1 + 2 + \cdots + k + (k+1) && \text{by definition} \\
&= S(k) + (k+1) && \text{because } S(k) = 1 + 2 + \cdots + k \\
&= \tfrac{1}{2}k(k+1) + (k+1) && \text{by the induction hypothesis} \\
&= \tfrac{1}{2}(k+1)(k+2) && \text{by algebra}
\end{aligned}$$

**b.** *Basis:* Let $n = 0$. Then, $C(n) = 0$ by definition, and $\frac{1}{4}(n^4 + 2n^3 + n^2) = 0$. So $C(n) = \frac{1}{4}(n^4 + 2n^3 + n^2)$ when $n = 0$.

*Induction:* Assume true for $n = k$ where $k \geq 0$ and prove true for $n = k + 1$. We can use this series of equalities:

$$\begin{aligned}
C(k+1) &= 1^3 + 2^3 + \cdots + k^3 + (k+1)^3 && \text{by definition} \\
&= C(k) + (k+1)^3 && C(k) = 1^3 + \cdots + k^3 \\
&= \tfrac{1}{4}(n^4 + 2n^3 + n^2) + (k+1)^3 && \text{induction hypothesis} \\
&= \tfrac{1}{4}((n+1)^4 + 2(n+1)^3 + (n+1)^2) && \text{by algebra}
\end{aligned}$$

**0.13** Dividing by $(a - b)$ is illegal, because $a = b$ hence $a - b = 0$ and division by 0 is undefined.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning

# Chapter 1

**1.12** Observe that $D \subseteq \mathtt{b}^*\mathtt{a}^*$ because $D$ doesn't contain strings that have ab as a substring. Hence $D$ is generated by the regular expression $(\mathtt{aa})^*\mathtt{b}(\mathtt{bb})^*$. From this description, finding the DFA for $D$ is more easily done.

**1.14** **a.** Let $M'$ be the DFA $M$ with the accept and non-accept states swapped. We show that $M'$ recognizes the complement of $B$, where $B$ is the language recognized by $M$. Suppose $M'$ accepts $x$. If we run $M'$ on $x$ we end in an accept state of $M'$. Because $M$ and $M'$ have swapped accept/non-accept states, if we run $M$ on $x$, we would end in a non-accept state. Therefore, $x \notin B$. Similarly, if $x$ is not accepted by $M'$, then it would be accepted by $M$. So $M'$ accepts exactly those strings not accepted by $M$. Therefore, $M'$ recognizes the complement of $B$.

Since $B$ could be any arbitrary regular language and our construction shows how to build an automaton to recognize its complement, it follows that the complement of any regular language is also regular. Therefore, the class of regular languages is closed under complement.

**b.** Consider the NFA in Exercise 1.16(a). The string $a$ is accepted by this automaton. If we swap the accept and reject states, the string $a$ is still accepted. This shows that swapping the accept and non-accept states of an NFA doesn't necessarily yield a new NFA recognizing the complementary language. The class of languages recognized by NFAs is, however, closed under complement. This follows from the fact that the class of languages recognized by NFAs is precisely the class of languages recognized by DFAs which we know is closed under complement from part (a).

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

**a.** $1\Sigma^*0$

**b.** $\Sigma^*1\Sigma^*1\Sigma^*1\Sigma^*$

**c.** $\Sigma^*0101\Sigma^*$

**d.** $\Sigma\Sigma0\Sigma^*$

**e.** $(0 \cup 1\Sigma)(\Sigma\Sigma)^*$

**f.** $(0 \cup (10)^*)^*1^*$

**g.** $(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)$

**h.** $\Sigma^*0\Sigma^* \cup 1111\Sigma^* \cup 1 \cup \varepsilon$

**i.** $(1\Sigma)^*(1 \cup \varepsilon)$

**j.** $0^*(100 \cup 010 \cup 001 \cup 00)0^*$

**k.** $\varepsilon \cup 0$

**l.** $(1^*01^*01^*)^* \cup 0^*10^*10^*$



© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**m.** $\emptyset$
**n.** $\Sigma^+$

**1.20** **a.** ab, $\varepsilon$; ba, aba
**b.** ab, abab; $\varepsilon$, aabb
**c.** $\varepsilon$, aa; ab, aabb
**d.** $\varepsilon$, aaa; aa, b
**e.** aba, aabbaa; $\varepsilon$, abbb
**f.** aba, bab; $\varepsilon$, ababab
**g.** b, ab; $\varepsilon$, bb
**h.** ba, bba; b, $\varepsilon$

**1.21** In both parts we first add a new start state and a new accept state. Several solutions are possible, depending on the order states are removed.
**a.** Here we remove state 1 then state 2 and we obtain
$\mathtt{a}^*\mathtt{b}(\mathtt{a} \cup \mathtt{ba}^*\mathtt{b})^*$
**b.** Here we remove states 1, 2, then 3 and we obtain
$\varepsilon \cup ((\mathtt{a} \cup \mathtt{b})\mathtt{a}^*\mathtt{b}((\mathtt{b} \cup \mathtt{a}(\mathtt{a} \cup \mathtt{b}))\mathtt{a}^*\mathtt{b})^*(\varepsilon \cup \mathtt{a}))$

**1.22** **b.** $\mathtt{/\#(\#}^*(\mathtt{a} \cup \mathtt{b}) \cup \mathtt{/)}^*\mathtt{\#}^+\mathtt{/}$

**1.24** **a.** $q_1, q_1, q_1, q_1$; 000.
**b.** $q_1, q_2, q_2, q_2$; 111.
**c.** $q_1, q_1, q_2, q_1, q_2$; 0101.
**d.** $q_1, q_3$; 1.
**e.** $q_1, q_3, q_2, q_3, q_2$; 1111.
**f.** $q_1, q_3, q_2, q_1, q_3, q_2, q_1$; 110110.
**g.** $q_1$; $\varepsilon$.

**1.25** A *finite state transducer* is a 5-tuple $(Q, \Sigma, \Gamma, \delta, q_0)$, where
i) $Q$ is a finite set called the *states*,
ii) $\Sigma$ is a finite set called the *alphabet*,
iii) $\Gamma$ is a finite set called the *output alphabet*,
iv) $\delta\colon Q \times \Sigma \longrightarrow Q \times \Gamma$ is the *transition function*,
v) $q_0 \in Q$ is the *start state*.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a *finite state transducer*, $w = w_1 w_2 \cdots w_n$ be a string over $\Sigma$, and $v = v_1 v_2 \cdots v_n$ be a string over the $\Gamma$. Then $M$ *outputs* $v$ if a sequence of states $r_0, r_1, \ldots, r_n$ exists in Q with the following two conditions:
i) $r_0 = q_o$
ii) $\delta(r_i, w_{i+1}) = (r_{i+1}, v_{i+1})$ for $i = 0, \ldots, n-1$.

**1.26** **a.** $T_1 = (Q, \Sigma, \Gamma, \delta, q_1)$, where
i) $Q = \{q_1, q_2\}$,
ii) $\Sigma = \{0, 1, 2\}$,
iii) $\Gamma = \{0, 1\}$,
iv) $\delta$ is described as

| | 0 | 1 | 2 |
|---|---|---|---|
| $q_1$ | $(q_1,0)$ | $(q_1,0)$ | $(q_2,1)$ |
| $q_2$ | $(q_1,0)$ | $(q_2,1)$ | $(q_2,1)$ |

v) $q_1$ is the start state.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**b.** $T_2 = (Q, \Sigma, \Gamma, \delta, q_1)$, where

i) $Q = \{q_1, q_2, q_3\}$,

ii) $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,

iii) $\Gamma = \{\mathtt{0}, \mathtt{1}\}$,

iv) $\delta$ is described as

| | a | b |
|---|---|---|
| $q_1$ | $(q_2,\mathtt{1})$ | $(q_3,\mathtt{1})$ |
| $q_2$ | $(q_3,\mathtt{1})$ | $(q_1,\mathtt{0})$ |
| $q_3$ | $(q_1,\mathtt{0})$ | $(q_2,\mathtt{1})$ |

v) $q_1$ is the start state.

**1.29** **b.** Let $A_2 = \{www \mid w \in \{\mathtt{0},\mathtt{1}\}^*\}$. We show that $A_2$ is nonregular using the pumping lemma. Assume to the contrary that $A_2$ is regular. Let $p$ be the pumping length given by the pumping lemma. Let $s$ be the string $\mathtt{a}^p\mathtt{b}\mathtt{a}^p\mathtt{b}\mathtt{a}^p\mathtt{b}$. Because $s$ is a member of $A_2$ and $s$ has length more than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. However, condition 3 implies that $y$ must consist only of as, so $xyyz \notin A_2$ and one of the first two conditions is violated. Therefore $A_2$ is nonregular.

**1.30** The error is that $s = \mathtt{0}^p\mathtt{1}^p$ *can* be pumped. Let $s = xyz$, where $x = \mathtt{0}$, $y = \mathtt{0}$ and $z = \mathtt{0}^{p-2}\mathtt{1}^p$. The conditions are satisfied because

i) for any $i \geq 0$, $xy^iz = \mathtt{0}\mathtt{0}^i\mathtt{0}^{p-2}\mathtt{1}^p$ is in $\mathtt{0}^*\mathtt{1}^*$.

ii) $|y| = 1 > 0$, and

iii) $|xy| = 2 \leq p$.

**1.31** We construct a DFA which alternately simulates the DFAs for $A$ and $B$, one step at a time. The new DFA keeps track of which DFA is being simulated. Let $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be DFAs for $A$ and $B$. We construct the following DFA $M = (Q, \Sigma, \delta, s_0, F)$ for the perfect shuffle of $A$ and $B$.

i) $Q = Q_1 \times Q_2 \times \{1, 2\}$.

ii) For $q_1 \in Q_1$, $q_2 \in Q_2$, $b \in \{1, 2\}$, and $a \in \Sigma$:

$$\delta((q_1, q_2, b), a) = \begin{cases} (\delta_1(q_1, a), q_2, 2) & b = 1 \\ (q_1, \delta_1(q_2, a), 1) & b = 2. \end{cases}$$

iii) $s_0 = (s_1, s_2, 1)$.

iv) $F = \{(q_1, q_2, 1) \mid q_1 \in F_1 \text{ and } q_2 \in F_2\}$.

**1.32** We construct an NFA which simulates the DFAs for $A$ and $B$, nondeterministically switching back and forth from one to the other. Let $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be DFAs for $A$ and $B$. We construct the following NFA $N = (Q, \Sigma, \delta, s_0, F)$ for the shuffle of $A$ and $B$.

i) $Q = Q_1 \times Q_2$.

ii) For $q_1 \in Q_1$, $q_2 \in Q_2$, and $a \in \Sigma$:

$$\delta((q_1, q_2), a) = \{(\delta_1(q_1, a), q_2), (q_1, \delta_2(q_2, a))\}.$$

iii) $s_0 = (s_1, s_2)$.

iv) $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ and } q_2 \in F_2\}$.

**1.33** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes $A$. Then we construct NFA $N = (Q', \Sigma, \delta', q_0', F')$ recognizing *DROP-OUT*$(A)$. The idea behind the construction is that $N$ simulates $M$ on its input, nondeterministically guessing the point at which the

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



dropped out symbol occurs. At that point $N$ guesses the symbol to insert in that place, without reading any actual input symbol at that step. Afterwards, it continues to simulate $M$.

We implement this idea in $N$ by keeping two copies of $M$, called the top and bottom copies. The start state is the start state of the top copy. The accept states of $N$ are the accept states of the bottom copy. Each copy contains the edges that would occur in $M$. Additionally, include $\varepsilon$ edges from each state $q$ in the top copy to every state in the bottom copy that $q$ can reach.

We describe $N$ formally. The states in the top copy are written with a T and the bottom with a B, thus: $(\mathrm{T}, q)$ and $(\mathrm{B}, q)$.

i) $Q' = \{\mathrm{T}, \mathrm{B}\} \times Q$,
ii) $q_0' = (\mathrm{T}, q_0)$,
iii) $F' = \{\mathrm{B}\} \times F$,
iv) $\delta'((\mathrm{T}, q), a) = \begin{cases} \{(\mathrm{T}, \delta(q, a))\} & a \in \Sigma \\ \{(\mathrm{B}, \delta(q, b)) \mid b \in \Sigma\} & a = \varepsilon \end{cases}$

$$\delta'((\mathrm{B}, q), a) = \begin{cases} \{(\mathrm{B}, \delta(q, a))\} & a \in \Sigma \\ \emptyset & a = \varepsilon \end{cases}$$

**1.35** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes $A$. We construct a new DFA $M' = (Q, \Sigma, \delta, q_0, F')$ that recognizes $A/B$. Automata $M$ and $M'$ differ only in the sets of accept states. Let $F' = \{r \mid$ starting at $r$ and reading a string in $B$ we get to an accept state of $M\}$. Thus $M'$ accepts a string $w$ iff there is a string $x \in B$ where $M$ accepts $wx$. Hence $M'$ recognizes $A/B$.

**1.36** For any regular language $A$, let $M_1$ be the DFA recognizing it. We need to find a DFA that recognizes $A^R$. Since any NFA can be converted to an equivalent DFA, it suffices to find an NFA $M_2$ that recognizes $A^R$.

We keep all the states in $M_1$ and reverse the direction of all the arrows in $M_1$. We set the accept state of $M_2$ to be the start state in $M_1$. Also, we introduce a new state $q_0$ as the start state for $M_2$ which goes to every accept state in $M_1$ by an $\epsilon$-transition.

**1.39** The idea is that we start by comparing the most significant bit of the two rows. If the bit in the top row is bigger, we know that the string is in the language. The string does not belong to the language if the bit in the top row is smaller. If the bits on both rows are the same, we move on to the next most significant bit until a difference is found. We implement this idea with a DFA having states $q_0$, $q_1$, and $q_2$. State $q_0$ indicates the result is not yet determined. States $q_1$ and $q_2$ indicate the top row is known to be larger, or smaller, respectively. We start with $q_0$. If the top bit in the input string is bigger, it goes to $q_1$, the only accept state, and stays there till the end of the input string. If the top bit in the input string is smaller, it goes to $q_2$ and stays there till the end of the input string. Otherwise, it stays in state $q_0$.

**1.40** Assume language $E$ is regular. Use the pumping lemma to a get a pumping length $p$ satisfying the conditions of the pumping lemma. Set $s = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^p \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p$. Obviously, $s \in E$ and $|s| \geq p$. Thus, the pumping lemma implies that the string $s$ can be written as $xyz$ with $x = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^a$, $y = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^b$, $z = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^c \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p$, where $b \geq 1$ and $a + b + c = p$. However, the string $s' = xy^0z = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^{a+c} \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p \notin E$, since $a + c < p$. That contradicts the pumping lemma. Thus $E$ is not regular.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**1.41** For each $n \geq 1$, we build a DFA with the $n$ states $q_0, q_1, \ldots, q_{n-1}$ to count the number of consecutive a's modulo $n$ read so far. For each character a that is input, the counter increments by 1 and jumps to the next state in $M$. It accepts the string if and only if the machine stops at $q_0$. That means the length of the string consists of all a's and its length is a multiple of $n$.

More formally, the set of states of $M$ is $Q = \{q_0, q_1, \ldots, q_{n-1}\}$. The state $q_0$ is the start state and the only accept state. Define the transition function as: $\delta(q_i, \mathtt{a}) = q_j$ where $j = i + 1 \bmod n$.

**1.42** By simulating binary division, we create a DFA $M$ with $n$ states that recognizes $C_n$. $M$ has $n$ states which keep track of the $n$ possible remainders of the division process. The start state is the only accept state and corresponds to remainder 0.

The input string is fed into $M$ starting from the most significant bit. For each input bit, $M$ doubles the remainder that its current state records, and then adds the input bit. Its new state is the sum modulo $n$. We double the remainder because that corresponds to the left shift of the computed remainder in the long division algorithm. If an input string ends at the accept state (corresponding to remainder 0), the binary number has no remainder on division by $n$ and is therefore a member of $C_n$.

The formal definition of $M$ is $(\{q_0, \ldots, q_{n-1}\}, \{\mathtt{0}, \mathtt{1}\}, \delta, q_0, \{q_0\})$. For each $q_i \in Q$ and $b \in \{\mathtt{0}, \mathtt{1}\}$, define $\delta(q_i, b) = q_j$ where $j = (2i + b) \bmod n$.

**1.43** Use the same construction given in the proof of Theorem 1.39, which shows the equivalence of NFAs and DFAs. We need only change $F'$, the set of accept states of the new DFA. Here we let $F' = \mathcal{P}(F)$. The change means that the new DFA accepts only when *all* of the possible states of the all-NFA are accepting.

**1.44** Let $A_k = \Sigma^* \mathtt{0}^{k-1} \mathtt{0}^*$. A DFA with $k$ states $\{q_0, \ldots, q_{k-1}\}$ can recognize $A_k$. The start state is $q_0$. For each $i$ from 0 to $k-2$, state $q_i$ branches to $q_{i+1}$ on 0 and to $q_0$ on 1. State $q_{k-1}$ is the accept state and branches to itself on 0 and to $q_0$ on 1.

In any DFA with fewer than $k$ states, two of the $k$ strings 1, 10, ... , $\mathtt{1}\mathtt{0}^{k-1}$ must cause the machine to enter the same state, by the pigeon hole principle. But then, if we add to both of these strings enough 0s to cause the longer of these two strings to have exactly $k-1$ 0s, the two new strings will still cause the machine to enter the same state, but one of these strings is in $A_k$ and the other is not. Hence, the machine must fail to produce the correct accept/reject response on one of these strings.

**1.45** **b.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing $A$, where $A$ is some regular language. We construct $M' = (Q', \Sigma, \delta, q_0', F')$ recognizing $NOEXTEND(A)$ as follows:

i) $Q' = Q$
ii) $\delta' = \delta$
iii) $q_0' = q_0$
iv) $F' = \{q \mid q \in F$ and there is no path of length $\geq 1$ from $q$ to an accept state$\}$.

**1.47** To show that $\equiv_L$ is an equivalence relation we show it is reflexive, symmetric, and transitive. It is reflexive because no string can distinguish $x$ from itself and hence $x \equiv_L x$ for every $x$. It is symmetric because $x$ is distinguishable from $y$ whenever $y$ is distinguishable from $x$. It is transitive because if $w \equiv_L x$ and $x \equiv_L y$, then for each $z$, $wz \in L$ iff $xz \in L$ and $xz \in L$ iff $yz \in L$, hence $wz \in L$ iff $yz \in L$, and so $w \equiv_L y$.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**1.49** **a.** $F$ is not regular, because the nonregular language $\{\mathtt{ab}^n\mathtt{c}^n \mid n \geq 0\}$ is the same as $F \cap \mathtt{ab}^*\mathtt{c}^*$, and the regular languages are closed under intersection.

**b.** Language $F$ satisfies the conditions of the pumping lemma using pumping length 2. If $s \in F$ is of length 2 or more we show that it can be pumped by considering four cases, depending on the number of a's that $s$ contains.

i) If $s$ is of the form $\mathtt{b}^*\mathtt{c}^*$, let $x = \varepsilon$, $y$ be the first symbol of $s$, and let $z$ be the rest of $s$.

ii) If $s$ is of the form $\mathtt{ab}^*\mathtt{c}^*$, let $x = \varepsilon$, $y$ be the first symbol of $s$, and let $z$ be the rest of $s$.

iii) If $s$ is of the form $\mathtt{aab}^*\mathtt{c}^*$, let $x = \varepsilon$, $y$ be the first two symbols of $s$, and let $z$ be the rest of $s$.

iv) If $s$ is of the form $\mathtt{aaa}^*\mathtt{b}^*\mathtt{c}^*$, let $x = \varepsilon$, $y$ be the first symbol of $s$, and let $z$ be the rest of $s$.

In each case, the strings $xy^iz$ are members of $F$ for every $i \geq 0$. Hence $F$ satisfies the conditions of the pumping lemma.

**c.** The pumping lemma is not violated because it states only that regular languages satisfy the three conditions, and it doesn't state that nonregular languages fail to satisfy the three conditions.

**1.50** The objective of this problem is for the student to pay close attention to the exact formulation of the pumping lemma.

**c.** This language is that same as the language in in part (b), so the solution is the same.

**e.** The minimum pumping length is 1. The pumping length cannot be 0, as in part (b). Any string in $(\mathtt{01})^*$ of length 1 or more contains 01 and hence can be pumped by dividing it so that $x = \varepsilon$, $y = \mathtt{01}$, and $z$ is the rest.

**f.** The minimum pumping length is 1. The pumping length cannot be 0, as in part (b). The language has no strings of length 1 or more so 1 is a pumping length. (the conditions hold vacuously).

**g.** The minimum pumping length is 3. The string 00 is in the language and it cannot be pumped, so the minimum pumping length cannot be 2. Every string in the language of length 3 or more contains a 1 within the first 3 symbols so it can be pumped by letting $y$ be that 1 and letting $x$ be the symbols to the left of $y$ and $z$ be the symbols to the right of $y$.

**h.** The minimum pumping length is 4. The string 100 is in the language but it cannot be pumped (down), therefore 3 is too small to be a pumping length. Any string of length 4 or more in the language must be of the form $xyz$ where $x$ is 10, $y$ is in $\mathtt{11}^*\mathtt{0}$ and $z$ is in $(\mathtt{11}^*\mathtt{0})^*\mathtt{0}$, which satisfies all of the conditions of the pumping lemma.

**i.** The minimum pumping length is 5. The string 1011 is in the language and it cannot be pumped. Every string in the language of length 5 or more (there aren't any) can be pumped (vacuously).

**j.** The minimum pumping length is 1. It cannot be 0 as in part (b). Every other string can be pumped, so 1 is a pumping length.

**1.51** **a.** Assume $L = \{\mathtt{0}^n\mathtt{1}^m\mathtt{0}^n \mid m, n \geq 0\}$ is regular. Let $p$ be the pumping length given by the pumping lemma. The string $s = \mathtt{0}^p\mathtt{10}^p \in L$, and $|s| \geq p$. Thus the pumping lemma implies that $s$ can be divided as $xyz$ with $x = \mathtt{0}^a$, $y = \mathtt{0}^b$, $z = \mathtt{0}^c\mathtt{10}^p$, where $b \geq 1$ and $a + b + c = p$. However, the string $s' = xy^0z = \mathtt{0}^{a+c}\mathtt{10}^p \notin L$, since $a + c < p$. That contradicts the pumping lemma.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**c.** Assume $C = \{w \mid w \in \{0,1\}^* \text{ is a palindrome}\}$ is regular. Let $p$ be the pumping length given by the pumping lemma. The string $s = 0^p 1 0^p \in C$ and $|s| \geq p$. Follow the argument as in part (a). Hence $C$ isn't regular, so neither is its complement.

**1.52** One short solution is to observe that $\overline{Y} \cap 1^*\#1^* = \{1^n\#1^n \mid n \geq 0\}$. This language is clearly not regular, as may be shown using a straightforward application of the pumping lemma. However, if $Y$ were regular, this language would be regular, too, because the class of regular languages is closed under intersection and complementation. Hence $Y$ isn't regular.

Alternatively, we can show $Y$ isn't regular directly using the pumping lemma. Assume to the contrary that $Y$ is regular and obtain its pumping length $p$. Let $s = 1^{p!}\#1^{2p!}$. The pumping lemma says that $s = xyz$ satisfying the three conditions. By condition 3, $y$ appears among the left-hand 1s. Let $l = |y|$ and let $k = (p!/l)$. Observe that $k$ is an integer, because $l$ must be a divisor of $p!$. Therefore, adding $k$ copies of $y$ to $s$ will add $p!$ additional 1s to the left-hand 1s. Hence, $xy^{1+k}z = 1^{2p!}\#1^{2p!}$ which isn't a member of $Y$. But condition 1 of the pumping lemma states that this string is a member of $Y$, a contradiction.

**1.53** The language $D$ can be written alternatively as $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 \cup \varepsilon$, which is obviously regular.

**1.54** The NFA $N_k$ guesses when it has read an a that appears at most $k$ symbols from the end, then counts $k-1$ more symbols and enters an accept state. It has an initial state $q_0$ and additional states $q_1$ thru $q_k$. State $q_0$ has transitions on both a and b back to itself and on a to state $q_1$. For $1 \leq i \leq k-1$, state $q_i$ has transitions on a and b to state $q_{i+1}$. State $q_k$ is an accept state with no transition arrows coming out of it. More formally, NFA $N_k = (Q, \Sigma, \delta, q_0, F)$ where

i) $Q = \{q_0, \ldots, q_k\}$

ii) $\delta(q,c) = \begin{cases} \{q_0\} & q = q_0 \text{ and } c = \mathtt{a} \\ \{q_0, q_1\} & q = q_0 \text{ and } c = \mathtt{b} \\ \{q_{i+1}\} & q = q_i \text{ for } 1 \leq i < k \text{ and } c \in \Sigma \\ \emptyset & q = q_k \text{ or } c = \varepsilon \end{cases}$

iii) $F = \{q_k\}$.

**1.55** Let $M$ be a DFA. Say that $w$ *leads* to state $q$ if $M$ is in $q$ after reading $w$. Notice that if $w_1$ and $w_2$ lead to the same state, then $w_1 p$ and $w_2 p$ also lead to the same state, for all strings $p$.

Assume that $M$ recognizes $C_k$ with fewer than $2^k$ states, and derive a contradiction. There are $2^k$ different strings of length $k$. By the pigeonhole principle, two of these strings $w_1$ and $w_2$ lead to the same state of $M$.

Let $i$ be the index of the first bit on which $w_1$ and $w_2$ differ. Since $w_1$ and $w_2$ lead $M$ to the same state, $w_1\mathtt{b}^{i-1}$ and $w_2\mathtt{b}^{i-1}$ lead $M$ to the same state. This cannot be the case, however, since one of the strings should be rejected and the other accepted. Therefore, any two distinct $k$ bit strings lead to different states of $M$. Hence $M$ has at least $2^k$ states.

**1.60** **a.** We construct $M'$ from $M$ by converting each transition that is traversed on symbol $a \in \Sigma$ to a sequence of transitions that are traversed while reading string $f(a) \in \Gamma^*$. The formal construction follows.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



Let $M = (Q, \Sigma, \delta, q_0, F)$. For each $a \in \Sigma$ let $z^a = f(a)$ and let $k_a = |z^a|$. We write $z^a = z_1^a z_2^a \ldots z_{k_a}^a$ where $z_i^a \in \Gamma$. Construct $M' = (Q', \Gamma, \delta', q_0, F)$. $Q' = Q \cup \{q_i^a |\ q \in Q, a \in \Sigma, 1 \le i \le k_a\}$

For every $q \in Q$,

$$\delta'(q, b) = \begin{cases} \{r |\ \delta(q, a) = r \text{ and } z^a = \varepsilon\} & b = \varepsilon \\ \{q_1^a |\ b = z_1^a\} & b \neq \varepsilon \end{cases}$$

$$\delta'(q_i^a, b) = \begin{cases} \{q_{i+1}^a |\ b = z_{i+1}^a\} & 1 \le i < k_a \text{ and } b \neq \varepsilon \\ \{r |\ \delta(q, a) = r\} & i = k_a \text{ and } b = \varepsilon \end{cases}$$

**b.** The above construction shows that the class of regular languages is closed under homomorphism. To show that the class of non-regular languages is not closed under homomorphism, let $B = \{0^m 1^m |\ m \ge 0\}$ and let $f\colon \{0,1\} \longrightarrow \{2\}$ be the homomorphism where $f(0) = f(1) = 2$. We know that $B$ is a non-regular language but $f(B) = \{2^{2m} |\ m \ge 0\}$ is a regular language.

**1.61** **a.** $RC(A) = \{w_{i+1} \cdots w_n w_1 \cdots w_i |\ w = w_1 \cdots w_n \in A \text{ and } 1 \le i \le n\}$, because we can let $x = w_1 \cdots w_i$ and $y = w_{i+1} \cdots w_n$. Then $RC(RC(A))$ gives the same language because if $st = w_{i+1} \cdots w_n w_1 \cdots w_i$ for some strings $s$ and $t$, then $ts = w_{j+1} \cdots w_n w_1 \cdots w_j$ for some $j$ where $1 \le j \le n$.

**b.** Let $A$ be a regular language that is recognized by DFA $M$. We construct a NFA $N$ that recognizes $RC(A)$. In the idea behind the construction, $N$ guesses the cut point nondeterministically by starting $M$ at any one of its states. Then $N$ simulates $M$ on the input symbols it reads. If $N$ finds that $M$ is in one of its accept states then $N$ may nondeterministically reset $M$ back to it's start state prior to reading the next symbol. Finally, $N$ accepts its input if the simulation ends in the same state it started in, and exactly one reset occurred during the simulation. Here is the formal construction. Let $M = (Q, \Sigma, \delta, q_0, F)$ recognize $A$ and construct $N = (Q', \Sigma, \delta', r, F')$ to recognize $RC(A)$.

Set $Q' = (Q \times Q \times \{1, 2\}) \cup \{r\}$. State $(q, r, i)$ signifies that $N$ started the simulation in $M$'s state $q$, it is currently simulating $M$ in state $r$, and if $i = 1$ a reset hasn't yet occurred whereas if $i = 2$ then a reset has occurred.

Set $\delta'(r, \varepsilon) = \{(q, q, 1) |\ q \in Q\}$. This starts simulating $M$ in each of its states, nondeterministically.

Set $\delta'((q, r, i), a) = \{(q, \delta(r, a), i)\}$ for each $q, r \in Q$ and $i \in \{1, 2\}$. This continues the simulation.

Set $\delta'((q, r, 1), \varepsilon) = \{(q, q_0, 2)\}$ for $r \in F$ and $q \in Q$. This allows $N$ to reset the simulation to $q_0$ if $M$ hasn't yet been reset and $M$ is currently in an accept state.

We set $\delta'$ to $\emptyset$ if it is otherwise unset.

$F' = \{(q, q, 2) |\ q \in Q\}$.

**1.62** Assume to the contrary that $ADD$ is regular. Let $p$ be the pumping length given by the pumping lemma. Choose $s$ to be the string $1^p$=0+$1^p$, which is a member of $ADD$. Because $s$ has length greater than $p$, the pumping lemma guarantees that $s$ can be split into three pieces, $s = xyz$, satisfying the conditions of the lemma. By the third condition in the pumping lemma have that $|xy| \le p$, it follows that $y$ is $1^k$ for some $k \ge 1$. Then $xy^2z$ is the string $1^{p+k}$=0+$1^p$, which is not a member of $ADD$, violating the pumping lemma. Hence $ADD$ isn't regular.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**1.63** Let $A = \{2^k | \ k \geq 0\}$. Clearly $B_2(A) = \mathtt{10}^*$ is regular. Use the pumping lemma to show that $B_3(A)$ isn't regular. Get the pumping length $p$ and chose $s \in B_3(A)$ of length $p$ or more. We show $s$ cannot be pumped. Let $s = xyz$. For string $w$, write $(w)_3$ to be the number that $w$ represents in base 3. Then

$$\lim_{i\to\infty} \frac{(xy^iz)_3}{(xy^i)_3} = 3^{|z|} \quad \text{and} \quad \lim_{i\to\infty} \frac{(xy^{i+1}z)_3}{(xy^i)_3} = 3^{|y|+|z|}$$

so

$$\lim_{i\to\infty} \frac{(xy^{i+1}z)_3}{(xy^iz)_3} = 3^{|y|}.$$

Therefore for a sufficiently large $i$,

$$\frac{(xy^{i+1}z)_3}{(xy^iz)_3} = 3^{|y|} \pm \alpha$$

for some $\alpha < 1$. But this fraction is a ratio of two members of $A$ and is therefore a whole number. Hence $\alpha = 0$ and the ratio is a power of 3. But the ration of two members of $A$ also is a power of 2. No number greater than 1 can be both a power of 2 and of 3, a contradiction.

**1.64** Given an NFA $M$ recognizing $A$ we construct an NFA $N$ accepting $A_{\frac{1}{2}-}$ using the following idea. $M$ keeps track of two states in $N$ using two "fingers". As it reads each input symbol, $N$ uses one finger to simulate $M$ on that symbol. Simultaneously, $M$ uses the other finger to run $M$ backwards from an accept state on a guessed symbol. $N$ accepts whenever the forward simulation and the backward simulation are in the same state, that is, whenever the two fingers are together. At those points we are sure that $N$ has found a string where another string of the same length can be appended to yield a member of $A$, precisely the definition of $A_{\frac{1}{2}-}$.

In the formal construction, Exercise 1.11 allows us to assume for simplicity that the NFA $M$ recognizing $A$ has a single accept state. Let $M = (Q, \Sigma, \delta, q_0, q_{\text{accept}})$. Construct NFA $N = (Q', \Sigma, \delta', q_0', F')$ recognizing the first halves of the strings in $A$ as follows:

i) $Q' = Q \times Q$.
ii) For $q, r \in Q$ define
$\delta'((q, r), a) = \{(u, v) | \ u \in \delta(q, a)$ and $r \in \delta(v, b)$ for some $b \in \Sigma\}$.
iii) $q_0{}' = (q_0, q_{\text{accept}})$.
iv) $F' = \{(q, q) | \ q \in Q\}$.

**1.65** Let $A = \{\mathtt{0}^*\mathtt{\#1}^*\}$. Thus, $A_{\frac{1}{3}-\frac{1}{3}} \cap \{\mathtt{0}^*\mathtt{1}^*\} = \{\mathtt{0}^n\mathtt{1}^n | \ n \geq 0\}$. Regular sets are closed under intersection, and $\{\mathtt{0}^n\mathtt{1}^n | \ n \geq 0\}$ is not regular, so $A_{\frac{1}{3}-\frac{1}{3}}$ is not regular.

**1.66** If M has a synchronizing sequence, then for any pair of states $(p, q)$ there is a string $w_{p,q}$ such that $\delta(p, w_{p,q}) = \delta(q, w_{p,q}) = h$, where $h$ is the home state. Let us run two copies of $M$ starting at states $p$ and $q$ respectively. Consider the sequence of pairs of states $(u, v)$ that the two copies run through before reaching home state $h$. If some pair appears in the sequence twice, we can delete the substring of $w_{p,q}$ that takes the copies of $M$ from one occurrence of the pair to the other, and thus obtain a new $w_{p,q}$. We repeat the process until all pairs of states in the sequence are distinct. The number of distinct state pairs is $k^2$, so $|w_{p,q}| \leq k^2$.

Suppose we are running $k$ copies of $M$ and feeding in the same input string $s$. Each copy starts at a different state. If two copies end up at the same state after some step,

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



they will do exactly the same thing for the rest of input, so we can get rid of one of them. If $s$ is a synchronizing sequence, we will end up with one copy of $M$ after feeding in $s$. Now we will show how to construct a synchronizing sequence $s$ of length at most $k^3$.

i) Start with $s = \epsilon$. Start $k$ copies of $M$, one at each of its states. Repeat the following two steps until we are left with only a single copy of $M$.
ii) Pick two of $M$'s remaining copies ($M_p$ and $M_q$) that are now in states $p$ and $q$ after reading $s$.
iii) Redefine $s = sw_{p,q}$. After reading this new $s$, $M_p$ and $M_q$ will be in the same state, so we eliminate one of these copies.

At the end of the above procedure, $s$ brings all states of $M$ to a single state. Call that state $h$. Stages 2 and 3 are repeated $k-1$ times, because after each repetition we eliminate one copy of $M$. Therefore $|s| \leq (k-1)k^2 < k^3$.

**1.67** Let $C = \Sigma^* B \Sigma^*$. Then $C$ is the language of all strings that contain some member of $B$ as a substring, If $B$ is regular then $C$ is also regular. We know from the solution to Problem 1.14 that the complement of a regular language is regular, and so $\overline{C}$ is regular. It is the language of all strings that do not contain some member of $B$ as a substring. Note that $A$ *avoids* $B = A \cap \overline{C}$. Therefore $A$ *avoids* $B$ is regular because we showed that the intersection of two regular languages is regular.

**1.68** **a.** Any string that doesn't begin and end with 0 obviously cannot be a member of $A$. If string $w$ does begin and end with 0 then $w = 0u0$ for some string $u$. Hence $A = 0\Sigma^*0$ and therefore $A$ is regular.

**b.** Assume for contradiction that $B$ is regular. Use the pumping lemma to get the pumping length $p$. Letting $s = 0^p 1 0^p$ we have $s \in B$ and so we can divide up $s = xyz$ according to the conditions of the pumping lemma. By condition 3, $xy$ has only 0s, hence the string $xyyz$ is $0^l 1 0^p$ for some $l > p$. But then $0^l 1 0^p$ isn't equal to $0^k 1 u 0^k$ for any $u$ and $k$, because the left-hand part of the string requires $k = l$ and the right-hand part requires $k \leq p$. Both together are impossible, because $l > p$. That contradicts the pumping lemma and we conclude that $B$ isn't regular.

**1.69** **a.** Let $s$ be a string in $U$ whose length is shortest among all strings in $U$. Assume (for contradiction) that $|s| \geq \max(k_1, k_2)$. One or both of the DFAs accept $s$ because $s \in U$. Say it is $M_1$ that accepts $s$. Consider the states $q_0, q_1, \ldots, q_l$ that $M_1$ enters while reading $s$, where $l = |s|$. We have $l \geq k_1$, so $q_0, q_1, \ldots, q_l$ must repeat some state. Remove the portion of $s$ between the repeated state to yield a shorter string that $M_1$ accepts. That string is in $U$, a contradiction. Thus $|s| < \max(k_1, k_2)$.

**b.** Let $s$ be a string in $\overline{U}$ whose length is shortest among all strings in $\overline{U}$. Assume (for contradiction) that $|s| \geq k_1 k_2$. Both of the DFAs reject $s$ because $s \in \overline{U}$. Consider the states $q_0, q_1, \ldots, q_l$ and $r_0, r_1, \ldots, r_l$ that $M_1$ and $M_2$ enter respectively while reading $s$, where $l = |s|$. We have $l \geq k_1 k_2$, so in the sequence of ordered pairs $(q_0, r_0), (q_1, r_1), \ldots, (q_l, r_l)$, some pair must repeat. Remove the portion of $s$ between the repeated pair to yield a shorter string that both $M_1$ and $M_2$ reject. That shorter string is in $\overline{U}$, a contradiction. Thus $|s| < k_1 k_2$.

**1.70** A PDA $P$ that recognizes $\overline{C}$ operates by nondeterministically choosing one of three cases. In the first case, $P$ scans its input and accepts if it doesn't contain exactly two #s. In the second case, $P$ looks for a mismatch between the first two strings that are separated by #. It does so by reading its input while pushing those symbols onto the stack until it reads #. At that point $P$ continues reading input symbols and matching

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



them with symbols that are popped off the stack. If a mismatch occurs, or if the stack empties before $P$ reads the next #, or if $P$ reads the next # before the stack empties, then it accepts. In the third case, $P$ looks for a mismatch between the last two strings that are separated by #. It does so by reading its input until it reads # and the it continues reading input symbols while pushing those symbols onto the stack until it reads a second #. At that point $P$ continues reading input symbols and matching them with symbols that are popped off the stack. If a mismatch occurs, or if the stack empties before $P$ reaches the end of the input or if $P$ reaches the end of the input before the stack empties, then it accepts.

Alternatively, here is a CFG that generates $\overline{C}$.

$$\begin{aligned}
A &\to YDY\#Y \mid Y\#YDY \mid Y \mid Y\#Y \mid Y\#Y\#Y\#Z \\
D &\to XDX \mid 0E1 \mid 1E0 \\
E &\to XEX \mid \# \\
X &\to 0 \mid 1 \\
Y &\to XY \mid \varepsilon \\
Z &\to Y\#Z \mid \varepsilon
\end{aligned}$$

**1.71** **a.** Observe that $B = 1\Sigma^*1\Sigma^*$ and thus is clearly regular.

**b.** We show $C$ is nonregular using the pumping lemma. Assume $C$ is regular and let $p$ be its pumping length. Let $s = 1^p01^p$. The pumping lemma says that $s = xyz$ satisfying the three conditions. Condition three says that $y$ appears among the left-hand 1s. We pump down to obtain the string $xz$ which is not a member of $C$. Therefore $C$ doesn't satisfy the pumping lemma and hence isn't regular.

**1.72** **a.** Let $B = \{0110\}$. Then $CUT(B) = \{0110, 1010, 0011, 1100, 1001\}$ and $CUT(CUT(B)) = \{0110, 1010, 0011, 1100, 1001, 0101\}$.

**b.** Let $A$ be a regular language that is recognized by DFA $M$. We construct a NFA $N$ that recognizes $CUT(A)$. This construction is similar to the construction in the solution to Problem 1.61. Here, $N$ begins by nondeterministically guessing two states $q_1$ and $q_2$ in $M$. Then $N$ simulates $M$ on its input beginning at state $q_1$. Whenever the simulation reaches $q_2$, nondeterministically $N$ may switch to simulating $M$ at its start state $q_0$ and if it reaches state $q_1$, it again nondeterministically may switch to state $q_2$. At the end of the input, if $N$'s simulation has made both switches and is now in one of $M$'s accept states, it has completed reading an input $yxz$ where $M$ accepts $xyz$, and so it accepts.

**1.73** **a.** The idea here is to show that a DFA with fewer than $2^k$ states must fail to give the right answer on at least one string. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA with $m$ states. Fix the value of $k$ and let $W = \{w \mid w \in \Sigma^k\}$ be the set of strings of length $k$.

For each of the $2^k$ strings $w \in W$, let $q_w$ be the state that $A$ enters after it starts in state $q_0$ and then reads $w$. If $m < 2^k$ then two different strings $s$ and $t$ must exist in $W$ where $A$ enters the same state, i.e., $q_s = q_t$. The string $ss$ is in $WW$ so $A$ must enter an accepting state after reading $ss$. Similarly, $st \notin WW$ so $A$ must enter a rejecting string after reading $st$. But $q_s = q_t$, so $A$ enters the same state after reading $ss$ or $st$, a contradiction. Therefore $m \geq 2^k$.

**b.** We can give an NFA $N = (Q, \Sigma, \delta, c_0, F)$ with $4k + 4$ states that recognizes $\overline{WW}$. The NFA $N$ branches into two parts. One part accepts if the inputs length isn't $2k$. The other part accepts if the input contains two unequal symbols that are $k$ separated.

Formally, let $Q = \{c_0, \ldots, c_{2k+1}, r, y_1, \ldots, y_k, z_1, \ldots, z_k\}$ and let $F = \{c_0, \ldots, c_{2k-1}, c_{2k+1}, y_k, z_k\}$.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



For $0 \leq i \leq 2k$ and $a \in \Sigma$, set $\delta(c_i, a) = \{c_{i+1}\}$ and $\delta(c_{k+1}, a) = \{c_{k+1}\}$.
Additionally, $\delta(c_0, \varepsilon) = \{r\}$, $\delta(r, \mathtt{0}) = \{r, y_1\}$ and $\delta(r, \mathtt{1}) = \{r, z_1\}$.
Finally, for $0 \leq i < k-1$, set $\delta(y_i, a) = \{y_{i+1}\}$ and $\delta(z_i, a) = \{z_{i+1}\}$,
$\delta(y_{k-1}, \mathtt{1}) = \{y_k\}$, $\delta(y_k, a) = \{y_k\}$, and
$\delta(z_{k-1}, \mathtt{1}) = \{z_k\}$, $\delta(z_k, a) = \{z_k\}$.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning

# Chapter 2

**2.1** Here are the derivations (but not the parse trees).

**a.** $E \Rightarrow T \Rightarrow F \Rightarrow \mathtt{a}$

**b.** $E \Rightarrow E\mathtt{+}T \Rightarrow T\mathtt{+}T \Rightarrow F\mathtt{+}T \Rightarrow F\mathtt{+}F \Rightarrow \mathtt{a+}F \Rightarrow \mathtt{a+a}$

**c.** $E \Rightarrow E\mathtt{+}T \Rightarrow E\mathtt{+}T\mathtt{+}T \Rightarrow T\mathtt{+}T\mathtt{+}T \Rightarrow F\mathtt{+}T\mathtt{+}T \Rightarrow F\mathtt{+}F\mathtt{+}T \Rightarrow F\mathtt{+}F\mathtt{+}F \Rightarrow \mathtt{a+}F\mathtt{+}F \Rightarrow \mathtt{a+a+}F \Rightarrow \mathtt{a+a+a}$

**d.** $E \Rightarrow T \Rightarrow F \Rightarrow \mathtt{(}E\mathtt{)} \Rightarrow \mathtt{(}T\mathtt{)} \Rightarrow \mathtt{(}F\mathtt{)} \Rightarrow \mathtt{((}E\mathtt{))} \Rightarrow \mathtt{((}T\mathtt{))} \Rightarrow \mathtt{((}F\mathtt{))} \Rightarrow \mathtt{((a))}$

**2.2** **a.** The following grammar generates $A$:

$$\begin{aligned} S &\rightarrow RT \\ R &\rightarrow \mathtt{a}R \mid \varepsilon \\ T &\rightarrow \mathtt{b}T\mathtt{c} \mid \varepsilon \end{aligned}$$

The following grammar generates $B$:

$$\begin{aligned} S &\rightarrow TR \\ T &\rightarrow \mathtt{a}T\mathtt{b} \mid \varepsilon \\ R &\rightarrow \mathtt{c}R \mid \varepsilon \end{aligned}$$

Both $A$ and $B$ are context-free languages and $A \cap B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. We know from Example 2.36 that this language is not context free. We have found two CFLs whose intersection is not context free. Therefore the class of context-free languages is not closed under intersection.

**b.** First, the context-free languages are closed under the union operation. Let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ be two arbitrary context free grammars. We construct a grammar $G$ that recognizes their union. Formally, $G = (V, \Sigma, R, S)$ where:

i) $V = V_1 \cup V_2$

ii) $R = R_1 \cup R_2 \cup \{S \rightarrow S_1,\ S \rightarrow S_2\}$
(Here we assume that $R_1$ and $R_2$ are disjoint, otherwise we change the variable names to ensure disjointness)

Next, we show that the CFLs are not closed under complementation. Assume, for a contradiction, that the CFLs are closed under complementation. Then, if $G_1$ and $G_2$ are context free grammars, it would follow that $\overline{L(G_1)}$ and $\overline{L(G_2)}$ are context free. We previously showed that context-free languages are closed under union and so $\overline{L(G_1)} \cup \overline{L(G_1)}$ is context free. That, by our assumption, implies that $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}}$ is context free. But by DeMorgan's laws, $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}} = L(G_1) \cap L(G_2)$. However, if $G_1$ and



© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



$G_2$ are chosen as in part (a), $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}}$ isn't context free. This contradiction shows that the context-free languages are not closed under complementation.

**2.4** **b.** $S \to 0R0 \mid 1R1 \mid \varepsilon$
$R \to 0R \mid 1R \mid \varepsilon$
**c.** $S \to 0 \mid 1 \mid 00S \mid 01S \mid 10S \mid 11S$
**e.** $S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$
**f.** $S \to S$

**2.5** **a.** This language is regular, so the PDA doesn't need to use its stack. The PDA reads the input and uses its finite control to maintain a counter which counts up to 3. It keeps track of the number of 1s it has seen using this counter. The PDA enters an accept state and scans to the end of the input if it has read three 1s.
**b.** This language is regular. The PDA reads the input and keeps track of the first and last symbol in its finite control. If they are the same, it accepts, otherwise it rejects.
**c.** This language is regular. The PDA reads the input and keeps track of the length (modulo 2) using its finite control. If the length is 1 (modulo 2) it accepts, otherwise it rejects.
**d.** The PDA reads the input and pushes the symbols onto the stack. At some point it nondeterministically guesses where the middle is. It looks at the middle symbol. If that symbol is a 1, it rejects. If it is a 0 the PDA reads the rest of the string, and for each character read, it pops one element off of its stack. If the stack is empty when it finishes reading the input, it accepts. If the stack is empty before it reaches the end of the input, or nonempty when the input is finished, it rejects.
**e.** The PDA reads the input and pushes each symbol onto its stack. At some point it nondeterministically guesses when it has reached the middle. It also nondeterministically guesses whether string has odd length or even length. If it guesses even, it pushes the current symbol it's reading onto the stack. If it guesses the string has odd length, it goes to the next input symbol without changing the stack. Then it reads the rest of the input, and it compares each symbol it reads to the symbol on the top of the stack. If they are the same, it pops the stack, and continues reading. If they are different, it rejects. If the stack is empty when it finishes reading the input, it accepts. If the stack is empty before it reaches the end of the input, or nonempty when the input is finished, it rejects.
**f.** The PDA never enters an accept state.

**2.6** **b.**

$$\begin{aligned} S &\to X\mathtt{b}X\mathtt{a}X \mid T \mid U \\ T &\to \mathtt{a}T\mathtt{b} \mid T\mathtt{b} \mid \mathtt{b} \\ U &\to \mathtt{a}U\mathtt{b} \mid \mathtt{a}U \mid \mathtt{a} \\ X &\to \mathtt{a}X \mid \mathtt{b}X \mid \varepsilon \end{aligned}$$

**d.**

$$\begin{aligned} S &\to M\#P\#M \mid P\#M \mid M\#P \mid P \\ P &\to \mathtt{a}P\mathtt{a} \mid \mathtt{b}P\mathtt{b} \mid \mathtt{a} \mid \mathtt{b} \mid \varepsilon \mid \# \mid \#M\# \\ M &\to \mathtt{a}M \mid \mathtt{b}M \mid \#M \mid \varepsilon \end{aligned}$$

Note that we need to allow for the case when $i = j$, that is, some $x_i$ is a palindrome. Also, $\varepsilon$ is in the language since it's a palindrome.

**2.9** A CFG $G$ that generates $A$ is given as follows:

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



$G = (V, \Sigma, R, S)$, $V = \{S, E_{\mathtt{ab}}, E_{\mathtt{bc}}, C, A\}$, and $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. The rules are:

$$\begin{aligned} S &\to E_{\mathtt{ab}}C \mid AE_{\mathtt{bc}} \\ E_{\mathtt{ab}} &\to \mathtt{a}E_{\mathtt{ab}}\mathtt{b} \mid \varepsilon \\ E_{\mathtt{bc}} &\to \mathtt{b}E_{\mathtt{bc}}\mathtt{c} \mid \varepsilon \\ C &\to C\mathtt{c} \mid \varepsilon \\ A &\to A\mathtt{a} \mid \varepsilon \end{aligned}$$

Initially substituting $E_{\mathtt{ab}}C$ for $S$ generates any string with an equal number of a's and b's followed by any number of c's. Initially substituting $E_{\mathtt{bc}}$ for $S$ generates any string with an equal number of b's and c's prepended by any number of a's.

The grammar is ambiguous. Consider the string $\varepsilon$. On the one hand, it can be derived by choosing $E_{\mathtt{ab}}$C with each of $E_{\mathtt{ab}}$ and $C$ yielding $\varepsilon$. On the other hand, $\varepsilon$ can be derived by choosing $AE_{\mathtt{bc}}$ with each of $A$ and $E_{\mathtt{bc}}$ yielding $\varepsilon$. In general, any string $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$ with $i = j = k$ can be derived ambiguously in this grammar.

**2.10**

1. Nondeterministically branch to either Stage 2 or Stage 6.
2. Read and push a's.
3. Read b's, while popping a's.
4. If b's finish when stack is empty, skip c's on input and *accept*.
5. Skip a's on input.
6. Read and push b's.
7. Read c's, while popping b's.
8. If c's finish when stack is empty, *accept*.

**2.11**

1. Place \$ and $E$ on the stack.
2. Repeat the following steps forever.
3. If the top of stack is the variable $E$, pop it and nondeterministically push either $E$+$T$ or $T$ into the stack.
4. If the top of stack is the variable $T$, pop it and nondeterministically push either $T$×$F$ or $F$ into the stack.
5. If the top of stack is the variable $F$, pop it and nondeterministically push either (*E*) or a into the stack.
6. If the top of stack is a terminal symbol, read the next symbol from the input and compare it to the terminal in the stack. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
7. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

**2.12** Informal description of a PDA that recognizes the CFG in Exercise 2.3:

1. Place the marker symbol \$ and the start variable $R$ on the stack.
2. Repeat the following steps forever.
3. If the top of stack is the variable $R$, pop it and nondeterministically push either $XRX$ or $S$ into the stack.
4. If the top of stack is the variable $S$, pop it and nondeterministically push either a$T$b or b$T$a into the stack.
5. If the top of stack is the variable $T$, pop it and nondeterministically push either $XTX$, $X$ or $\varepsilon$ into the stack.
6. If the top of stack is the variable $X$, pop it and nondeterministically push either a or b into the stack.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



7. If the top of stack is a terminal symbol, read the next symbol from the input and compare it to the terminal symbol in the stack. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
8. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

**2.13** **a.** $L(G)$ is the language of strings of 0s and #s that either contain exactly two #s and any number of 0s, or contain exactly one # and the number of 0s on the right-hand side of the # is twice the number of 0s on the left-hand side of the #.

**b.** Assume $L(G)$ is regular and obtain a contradiction. Let $A = L(G) \cap 0^*\#0^*$. If $L(G)$ is regular, so is $A$. But we can show $A = \{0^k\#0^{2k} \mid k \geq 0\}$ is not regular by using a standard pumping lemma argument.

**2.14**

$$\begin{aligned} S_0 &\to AB \mid CC \mid BA \mid BD \mid BB \mid \varepsilon \\ A &\to AB \mid CC \mid BA \mid BD \mid BB \\ B &\to CC \\ C &\to 0 \\ D &\to AB \end{aligned}$$

**2.16** Let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ be CFGs.

Construct a new CFG $G_\cup$ where $L(G_\cup) = L(G_1) \cup L(G_2)$. Let $S$ be a new variable that is neither in $V_1$ nor in $V_2$ and assume these sets are disjoint.

Let $G_\cup = (V_1 \cup V_2 \cup \{S\},\ \Sigma,\ R_1 \cup R_2 \cup \{r_0\},\ S)$, where $r_0$ is $S \to S_1 \mid S_2$.

We construct CFG $G_\circ$ that generates $L(G_1) \circ L(G_2)$ as in $G_\cup$ by changing $r_0$ in $G_\cup$ into $S \to S_1S_2$.

To construct CFG $G_*$ that generates the language $L(G_1)^*$, let $S'$ be a new variable not in $V_1$ and make it the starting variable. Let $r_0$ be $S' \to S'S_1 \mid \varepsilon$ be a new rule in $G_*$.

**2.17** Let $A$ be a regular language generated by regular expression $R$. If $R$ is one of the atomic regular expressions $b$, for $b \in \Sigma_\varepsilon$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to b\}, S)$. If $R$ is the atomic regular expressions $\emptyset$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to S\}, S)$. If $R$ is a regular expression composed of smaller regular expressions combined with a regular operation, use the result of Problem 2.16 to construct an equivalent CFG out of the CFGs that are equivalent to the smaller expressions.

**2.18** $S$ can generate a string of $T$s. Each $T$ can generate strings in $\{\mathtt{a}^m b^m \mid m \geq 1\}$. Here are two different leftmost derivations of ababab.

$S \Rightarrow SS \Rightarrow SSS \Rightarrow TSS \Rightarrow \mathtt{ab}SS \Rightarrow \mathtt{ab}TS \Rightarrow \mathtt{abab}S \Rightarrow \mathtt{abab}T \Rightarrow \mathtt{ababab}$.

$S \Rightarrow SS \Rightarrow TS \Rightarrow \mathtt{ab}S \Rightarrow \mathtt{ab}SS \Rightarrow \mathtt{ab}TS \Rightarrow \mathtt{abab}S \Rightarrow \mathtt{abab}T \Rightarrow \mathtt{ababab}$.

The ambiguity arises because $S$ can generate a string of $T$s in multiple ways. We can prevent this behavior by forcing $S$ to generate each string of $T$s in a single way by changing the first rule to be $S \to TS$ instead of $S \to SS$. In the modified grammar, a leftmost derivation will repeatedly expand $T$ until it is eliminated before expanding $S$, and no other option for expanding variables is possible, so only one leftmost derivation is possible for each generated string.

**2.19** Let $A$ be a CFG that is recognized by PDA $P$ and construct the following PDA $P'$ that recognizes $RC(A)$. For simplicity, assume that $P$ empties its stack when it accepts its

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



input. The new PDA $P'$ must accept input $yx$ when $P$ accepts $xy$. Intuitively, would like $P'$ to start by guessing the state and the stack contents that $P$ would be in after reading $x$, so that $P'$ can simulate $P$ on $y$, and then if $P$ ends up in an accept state after reading $y$, we can simulate $P$ on $x$ to check that it ends up with the initially guessed start and stack contents. However, $P'$ has no way to store the initially guessed stack contents to check that it matches the stack after reading $x$. To avoid this difficulty, $P'$ guesses these stack symbols only at the point when $P$ would be popping them while reading $y$. It records these guesses by pushing a specially marked copy of each guessed symbol. When $P'$ guesses that it has finished reading $y$, it will have a copy of the stack that $P$ would have when it finishes reading $x$ but in reverse order, with marked symbols instead of regular symbols. Then, while $P'$ simulates $P$ reading $x$, whenever $P$ pushes a symbol, $P'$ may guess that it is one of the symbols that would have been popped while $P$ read $y$, and if so $P'$ pops the stack and matches the symbol $P$ would push with the marked symbol $P'$ popped.

**2.20** Let $B = \{1^m 2^n 3^n 4^m \mid n, m \geq 0\}$ which is a CFL. If $CUT(B)$ were a CFL then the language $CUT(B) \cap 2^*1^*3^*4^*$ would be a CFL because Problem 2.30 shows that the intersection of a CFL and a regular language is a CFL. But that intersection is $\{2^n 1^m 3^n 4^m \mid n, m \geq 0\}$ which is easily shown to be a non-CFL using the pumping lemma.

**2.21** We show that an ambiguous CFG cannot be a DCFG. If $G$ is ambiguous, $G$ derives some string $s$ with at least two different parse trees and therefore $s$ has at least two different rightmost derivations and at least two different leftmost reductions. Compare the steps of two of these different leftmost reductions and locate the first string where these reduction differ. The preceding string must be the same in both reductions, but it must have two different handles. Hence $G$ is not a DCFG.

**2.23** **a.** Let $B = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \neq j \text{ for } i, j, k \geq 0\}$ and $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid j \neq k \text{ for } i, j, k \geq 0\}$. The following DPDA recognizes $B$. Read all a's (if any) and push them onto the stack. Read b's (if any) while popping the a's. If the b's finish while a's remain on the stack or if b's remain unread when the stack becomes empty, then scan to the end of the input and *accept*, assuming that the input is in $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$ which was checked in parallel. Otherwise, *reject*.

A similar DPDA recognizes $C$. Thus $B$ and $C$ are DCFLs. However we have shown in the text that $A = B \cup C$ isn't a DCFL. Therefore the class of DCFLs is not closed under union.

**b.** We have shown in the text that the class of DCFLs is closed under complement. Thus, if the class of DCFLs were closed under intersection, the class would also be closed under union because $B \cup C = \overline{B} \cap \overline{C}$, contradicting Part **a** of this problem. Hence the class of DCFLs is not closed under intersection.

**c.** Define $B$ and $C$ as in Part **a**. Let $B' = (\mathtt{01} \cup \mathtt{0})B$ and $C' = (\mathtt{10} \cup \mathtt{1})C$. Then $D = B' \cup C'$ is a DCFL because a DPDA can determine which test to use by reading the first few symbols of the input. Let $E = (\mathtt{1} \cup \varepsilon)D$ which is $(\mathtt{101} \cup \mathtt{10} \cup \mathtt{01} \cup \mathtt{0})B \cup (\mathtt{110} \cup \mathtt{11} \cup \mathtt{10} \cup \mathtt{1})C$. To see that $E$ isn't a DCFL, take $F = E \cap \mathtt{01}(\mathtt{abc})^*$. The intersection of a DCFL and a regular language is a DCFL so if $E$ were a DCFL then $F$ would be a DCFL but $F = \mathtt{01}A$ which isn't a DCFL for the same reason $A$ isn't a DCFL.

**d.** Define $C$ and $F$ as in Part **c**. The language $H = \mathtt{1} \cup \varepsilon \cup D$ is easily seen to be a DCFL. However $H^*$ is not a DCFL because $H^* \cap \mathtt{01}(\mathtt{abc})^* = F \cup \mathtt{1} \cup \varepsilon$ and the latter isn't a DCFL for the same reason that $F$ isn't a DCFL.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**e.** Define $B$ and $C$ as in Part **a**. Let $K = \mathtt{0}B^{\mathcal{R}} \cup \mathtt{1}C^{\mathcal{R}}$. First, show that $K$ is a DCFL by giving a DPDA which operates as follows. After reading the first symbol, the DPDA follows the deterministic procedure to recognize membership in $B^{\mathcal{R}}$ or in $C^{\mathcal{R}}$ depending on whether the first symbol is 0 or 1. Next, show that $K^{\mathcal{R}}$ is not a DCFL by showing how to convert a DPDA $P$ for $K^{\mathcal{R}}$ into a DPDA $P'$ for the language $A = B \cup C$ which the text shows isn't a DCFL. Modify $P$ so that it recognizes the endmarked version of $K^{\mathcal{R}}$. Thus $L(P) = K^{\mathcal{R}}\dashv = B\mathtt{0}\dashv \cup C\mathtt{1}\dashv$. It is enough to design $P'$ to recognize $A\dashv = B\dashv \cup C\dashv$ because $A\dashv$ is a DCFL iff $A$ is a DCFL. It simulates $P$ but also keeps additional information on the stack which says what $P$ would do if its next input symbol were 0 or it it were 1. Then when $P'$ reads $\dashv$, it can use this information to accept when $P$ would have accepted if the prior input symbol had been either a 0 or a 1.

**2.24** **a.** Let $E = \{w \mid w \text{ has an equal number of } \mathtt{a}\text{'s and } \mathtt{b}\text{'s}\}$. It is enough to show that $T \overset{*}{\Rightarrow} w$ iff $w \in E$. The forward direction is straightforward. If $T \overset{*}{\Rightarrow} w$ then $w \in E$ because every substitution adds an equal number of a's and b's. The reverse direction is the following claim.

*Claim:* If $w \in E$ then $T \overset{*}{\Rightarrow} w$.

Proof by induction on $|w|$, the length of $w$.

Basis, $|w| = 0$: Then $w = \varepsilon$ and the rule $T \to \varepsilon$ shows that $T \overset{*}{\Rightarrow} w$.

Induction step: Let $k > 0$ and assume the claim is true whenever $|w| < k$. Prove the claim is true for $|w| = k$.

Take $w = w_1 \cdots w_k \in E$ where $|w| = k$ and each $w_i$ is a or b. Let $x = w^{\mathcal{R}}$. For $1 \le i \le k$, let $a_i$ ($b_i$) be the number of a's (b's) that appear among the first $i$ symbols of $x$ and let $c_i = a_i - b_i$. In other words, $c_i$ is the running count of the excess number of a's over b's across $x$. Because $w \in E$ and $E$ is closed under reversal, we know that $c_k = 0$. Let $m$ be the smallest $i$ for which $c_i = 0$. Consider the case where $x_1 = \mathtt{a}$. Then $c_1 = 1$. Moreover, $c_i > 0$ from $i = 1$ to $m - 1$ and $c_{m-1} = 1$. Thus, $x_2 \cdots x_{m-1} \in E$ and $x_m = \mathtt{b}$ and then also $x_{m+1} \cdots x_k \in E$. We can write $x = \mathtt{a}x_2 \cdots x_{m-1}\mathtt{b}x_{m+1} \cdots x_k$ and so $w = (x_{m-1} \cdots x_k)^{\mathcal{R}}\mathtt{b}(x_2 \cdots x_{m-1})^{\mathcal{R}}\mathtt{a}$. The induction assumption implies that $T \overset{*}{\Rightarrow} (x_{m-1} \cdots x_k)^{\mathcal{R}}$ and $T \overset{*}{\Rightarrow} (x_2 \cdots x_{m-1})^{\mathcal{R}}$. Therefore the rule $T \to T\mathtt{b}T\mathtt{a}$ shows that $T \overset{*}{\Rightarrow} w$. A similar argument works for the case where $x_1 = \mathtt{b}$.

**b.** Omitted

**c.** The DPDA reads the input and performs the following actions for each symbol read. If the stack is empty or if the input symbol is the same as the top stack symbol, the DPDA pushes the input symbol. Otherwise, if the top stack symbol differs from the input symbol, it pops the stack. When it reads $\dashv$, it accepts if the stack is empty, and otherwise it doesn't accept.

**2.26** Following the hint, the modified $P$ would accept the strings of the form $\{\mathtt{a}^m\mathtt{b}^m\mathtt{c}^m \mid m \ge 1\}$. But that is impossible because this language is not a CFL.

**2.27** Assume that DPDA $P$ recognizes $B$ and construct a modified DPDA $P'$ which simulates $P$ while reading a's and b's. If $P$ enters an accept state, $P'$ checks whether the next input symbol is a c, and if so it simulates $P$ on bc (pretending it has read an extra b) and then continues to simulate $P$ on the rest of the input. It accepts only when $P$ enters an accept state after reading c's. Then $P'$ recognizes the non-CFL $\{\mathtt{a}^m\mathtt{b}^m\mathtt{c}^m \mid m \ge 1\}$, an impossibility.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**2.28** Assume to the contrary that DPDA $P$ recognizes $C$. For a state $q$ and a stack symbol $x$, call $(q, x)$ a *minimal pair* if when $P$ is started $q$ with $x$ on the top of its stack, $P$ never pops its stack below $x$, no matter what input string $P$ reads from that point on. In that case, the contents of $P$'s stack at that point cannot affect its subsequent behavior, so $P$'s subsequent behavior can depend only on $q$ and $x$. Additionally, call $(q, \varepsilon)$ a minimal pair. It corresponds to starting $P$ in state $q$ with an empty stack.

*Claim:* Let $y$ be any input to $P$. Then $y$ can be extended to $z = ys$ for some $s \in \{0,1\}^*$ where $P$ on $z$ enters a minimal pair.

To prove this claim, observe that if $P$ on input $y$ enters a minimal pair then we are done immediately, and if $P$ on input $y$ enters a pair $(q, x)$ which isn't minimal then some input $s_1$ exists on which $P$ pops its stack below $x$. If $P$ on input $ys_1$ then enters a minimal pair, we are again done because we can let $z = ys_1$. If that pair still isn't minimal then some input $s_2$ exists which take $P$ to an even lower stack level. If $P$ on $ys_1s_2$ enters a minimal pair, we are done because we can let $z = ys_1s_2$. This procedure must terminate with a minimal pair, because $P$'s stack shrinks at every step and will eventually become empty. Thus, the claim is proved.

Let $k = 1 + |Q| \times (|\Gamma| + 1)$ be any value which is greater than the total number of minimal pairs. For every $i \leq k$, let $y_i = 10^i1$. The strings $y_i$ are all distinct, and no string $y_i$ is a prefix of some other string $y_j$. The claim shows that we can extend these to $z_1, \ldots, z_k \in \{0,1\}^*$ where each $z_i$ takes $P$ to a minimal pair. Because $k$ exceeds the number of minimal pairs, two of these strings, $z_i$ and $z_j$, lead to the same minimal pair. Observe that $P$ accepts both $z_iz_i^{\mathcal{R}}$ and $z_jz_j^{\mathcal{R}}$ because these strings are members of $C$. But because $z_i$ and $z_j$ lead to the same minimal pair, $P$'s will behave identically if we append the same string to either of them. Thus $P$ accepts input $z_iz_j^{\mathcal{R}}$ because it accepts input $z_jz_j^{\mathcal{R}}$. But $z_iz_j^{\mathcal{R}} \notin C$, a contradiction.

**2.31** The grammar generates all strings not of the form $\mathtt{a}^k\mathtt{b}^k$ for $k \geq 0$. Thus the complement of the language generated is $\overline{L(G)} = \{\mathtt{a}^k\mathtt{b}^k \mid k \geq 0\}$. The following grammar generates $\overline{L(G)}$: $\{\{S\}, \{\mathtt{a}, \mathtt{b}\}, \{S \to \mathtt{a}S\mathtt{b} \mid \varepsilon\}, S\}$.

**2.32** Let $P$ be a PDA that recognizes $A$. We construct a PDA $P'$ that recognizes $A/B$ as follows. It operates by reading symbols and simulating $P$. Whenever $P$ branches nondeterministically, so does $P'$. In addition, at every point $P'$ nondeterministically guesses that it has reached the end of the input and refrains from reading any more input symbols. Instead, it guesses additional symbols as if they were input and continues to simulate $P$ on this guessed input, while at the same time using its finite control to simulate a DFA for $B$ on the guessed input string. If $P'$ simultaneously reaches accept states in both of these simulations, $P'$ enters an accept state.

**2.33** The following CFG $G$ generates the language $C$ of all strings with twice as many a's as b's. $S \to \mathtt{b}S\mathtt{aa} \mid \mathtt{aa}S\mathtt{b} \mid \mathtt{a}S\mathtt{b}S\mathtt{a} \mid SS \mid \varepsilon$.

Clearly $G$ generates only strings in $C$. We prove inductively that every string in $C$ is in $L(G)$. Let $s$ be a string in $C$ of length $k$.

*Basis* If $k = 0$ then $S = \varepsilon \in L(G)$.

*Induction step* If $k > 0$, assume all strings in $C$ of length less than $k$ are generated by $G$ and show $s$ is generated by $G$.

If $s = s_1 \ldots s_k$ then for each $i$ from 0 to $k$ let $c_i$ to be the number of a's minus twice the number of b's in $s_1 \ldots s_i$. We have $c_0 = 0$ and $c_k = 0$ because $s \in L(G)$. Next we consider two cases.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



i) If $c_i = 0$ for some $i$ besides 0 and $k$, we divide into two substrings $s = tu$ where $t$ is the first $i$ symbols and $u$ is the rest. Both $t$ and $u$ are members of $C$ and hence are in $L(G)$ by the induction hypothesis. Therefore the rule $S \to SS$ show that $s \in L(G)$.

ii) If $c_i \neq 0$ for all $i$ besides 0 and $k$, we consider three subcases.

   i. If $s$ begins with b, then all $c_i$ from $i = 1$ to $k-1$ are negative, because $c_1 = -2$ and jumping from a negative $c_i$ to a positive $c_{i+1}$ isn't possible because the $c$ values can increase only by 1 at each step. Hence $s$ ends with aa, because no other ending would give $c_k = 0$. Therefore $s = \texttt{btaa}$ where $t \in C$. By the induction hypothesis, $t \in L(G)$ and so the rule $S \to \texttt{b}S\texttt{aa}$ shows that $s \in L(G)$.

   ii. If $s$ begins with a and all $c_i$ are non-negative, then $s_2 = \texttt{a}$ and $s_k = \texttt{b}$. Therefore $s = \texttt{aatb}$ where $t \in C$. By the induction hypothesis, $t \in L(G)$ and so the rule $S \to \texttt{aa}S\texttt{b}$ shows that $s \in L(G)$.

   iii. If $s$ begins with a and some $c_i$ is negative, then select the lowest $i$ for which a negative $c_i$ occurs. Then $c_{i-1} = +1$ and $c_i = -1$, because the $c$ values can decrease only by 2 at each step. Hence $s_i = \texttt{b}$. Furthermore $s$ ends with a, because no other ending would give $c_k = 0$. If we let $t$ be $s_2 \cdots s_{i-1}$ and $u$ be $s_{i+1} \cdots s_{k-1}$ then $s = \texttt{a}t\texttt{b}u\texttt{a}$ where both $t$ and $u$ are members of $C$ and hence are in $L(G)$ by the induction hypothesis. Therefore the rule $S \to \texttt{a}S\texttt{b}S\texttt{a}$ shows that $s \in L(G)$.

**2.34** We construct a PDA $P$ that recognizes $C$. First it nondeterministically branches to check either of two cases: that $x$ and $y$ differ in length or that they have the same length but differ in some position. Handling the first case is straightforward. To handle the second case, it operates by guessing corresponding positions on which the strings $x$ and $y$ differ, as follows. It reads the input at the same time as it pushes some symbols, say 1s, onto the stack. At some point it nondeterministically guesses a position in $x$ and it records the symbol it is currently reading there in its finite memory and skips to the #. Then it pops the stack while reading symbols from the input until the stack is empty and checks that the symbol it is now currently reading is different from the symbol it had recorded. If so, it accepts.

Here is a more detailed description of $P$'s algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, $P$ rejects on that branch of the computation.

**1.** Nondeterministically jump to either 2 or 4.
**2.** Read and push these symbols until read #. *Reject* if # never found.
**3.** Read and pop symbols until the end of the tape. *Reject* if another # is read or if the stack empties at the same time the end of the input is reached. Otherwise *accept*.
**4.** Read next input symbol and push 1 onto stack.
**5.** Nondeterministically jump to either 4 or 6.
**6.** Record the current input symbol $a$ in the finite control.
**7.** Read input symbols until # is read.
**8.** Read the next symbol and pop the stack.
**9.** If stack is empty, go to 10, otherwise go to 8.
**10.** *Accept* if the current input symbol isn't $a$. Otherwise *reject*.

**2.35** We construct a PDA $P$ recognizing $D$. This PDA guesses corresponding places on which $x$ and $y$ differ. Checking that the places correspond is tricky. Doing so relies on the

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



observation that the two corresponding places are $n/2$ symbols apart, where $n$ is the length of the entire input. Hence, by ensuring that the number of symbols between the guessed places is equal to the number other symbols, the PDA can check that the guessed places do indeed correspond, Here we give a more detailed description of the PDA algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, $P$ rejects on that branch of the computation.

1. Read next input symbol and push 1 onto the stack.
2. Nondeterministically jump to either 1 or 3.
3. Record the current input symbol $a$ in the finite control.
4. Read next input symbol and pop the stack. Repeat until stack is empty.
5. Read next input symbol and push 1 onto the stack.
6. Nondeterministically jump to either 5 or 7.
7. *Reject* if current input symbol equals $a$.
8. Read next input symbol and pop the stack. Repeat until stack is empty.
9. *Accept* if input is empty.

Alternatively we can give a CFG for this language as follows.

$$\begin{aligned} S &\to AB \mid BA \\ A &\to XAX \mid \mathtt{0} \\ B &\to XBX \mid \mathtt{1} \\ X &\to \mathtt{0} \mid \mathtt{1} \end{aligned}$$

**2.38** Consider a derivation of $w$. Each application of a rule of the form $A \to BC$ increases the length of the string by 1. So we have $n-1$ steps here. Besides that, we need exactly $n$ applications of terminal rules $A \to a$ to convert the variables into terminals. Therefore, exactly $2n-1$ steps are required.

**2.39** **a.** To see that $G$ is ambiguous, note that the string

`if condition then if condition then a:=1 else a:=1`

has two different leftmost derivations (and hence parse trees):

1. ⟨STMT⟩
   → ⟨IF-THEN⟩
   → `if condition then` ⟨STMT⟩
   → `if condition then` ⟨IF-THEN-ELSE⟩
   → `if condition then if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
   → `if condition then if condition then a:=1 else` ⟨STMT⟩
   → `if condition then if condition then a:=1 else a:=1`

2. ⟨STMT⟩
   → ⟨IF-THEN-ELSE⟩
   → `if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
   → `if condition then` ⟨IF-THEN⟩ `else` ⟨STMT⟩
   → `if condition then if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
   → `if condition then if condition then a:=1 else` ⟨STMT⟩
   → `if condition then if condition then a:=1 else a:=1`

**b.** The ambiguity in part a) arises because the grammar allows matching an `else` both to the nearest and to the farthest `then`. To avoid this ambiguity we construct a new grammar that only permits the nearest match, by disabling derivations which introduce an ⟨IF-THEN⟩

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



before an `else`. This grammar has two new variables: ⟨E-STMT⟩ and ⟨E-IF-THEN-ELSE⟩, which work just like their non-⟨E⟩ counterparts except that they cannot generate the dangling ⟨IF-THEN⟩. The rules of the new grammar are the same as for the old grammar except we remove the ⟨IF-THEN-ELSE⟩ rule and add the following new rules:

$$\begin{aligned}
\langle\text{E-STMT}\rangle &\rightarrow \langle\text{ASSIGN}\rangle \mid \langle\text{E-IF-THEN-ELSE}\rangle \\
\langle\text{E-IF-THEN-ELSE}\rangle &\rightarrow \texttt{if condition then}\ \langle\text{E-STMT}\rangle\ \texttt{else}\ \langle\text{E-STMT}\rangle \\
\langle\text{IF-THEN-ELSE}\rangle &\rightarrow \texttt{if condition then}\ \langle\text{E-STMT}\rangle\ \texttt{else}\ \langle\text{STMT}\rangle
\end{aligned}$$

**2.40** I found these problems to be surprisingly hard.

**a.** Define an *a-string* to be a string where every prefix has at least as many a's as b's. The following grammar $G$ generates all a-strings unambiguously.

$$\begin{aligned}
M &\rightarrow A\mathtt{a}M \mid A \\
A &\rightarrow \mathtt{a}A\mathtt{b}A \mid \varepsilon
\end{aligned}$$

Here is a proof sketch. First we claim that $A$ generates all balanced a-strings unambiguously. Let $w = w_1 \cdots w_n$ be any string. Let $c_i$ be the number of a's minus the number of b's in positions 1 through $i$ in $w$. The *mate* of a at position $i$ in $w$ is the b at the lowest position $j > i$ where $c_j < c_i$. It is easy to show inductively that for any balanced a-string $w$ and any parse tree for $w$ generated from $A$, the rule $A \rightarrow \mathtt{a}A\mathtt{b}A$ generates the mated pairs at the same time, hence it divides the string in a unique way and consequently grammar is unambiguous.

Next we claim that $M$ generates all a-strings that have an excess of a's. Say that a is *unmated* if it has no mate. We can show inductively that for any a-string, the $M$ rule $M \rightarrow A\mathtt{a}M$ generates the unmated a's and the $A$ rules generates the mated pairs. The generation can be done in only one way so the grammar is unambiguous.

**b.**

$$\begin{aligned}
E &\rightarrow \mathtt{a}A\mathtt{b}E \mid \mathtt{b}B\mathtt{a}E \mid \varepsilon \\
A &\rightarrow \mathtt{a}A\mathtt{b}A \mid \varepsilon \\
B &\rightarrow \mathtt{b}B\mathtt{a}B \mid \varepsilon
\end{aligned}$$

Proof omitted.

**c.**

$$\begin{aligned}
E &\rightarrow \mathtt{a}A\mathtt{b}E \mid \mathtt{b}B\mathtt{a}E \mid F \\
A &\rightarrow \mathtt{a}A\mathtt{b}A \mid \varepsilon \\
B &\rightarrow \mathtt{b}B\mathtt{a}B \mid \varepsilon \\
F &\rightarrow \mathtt{a}AF \mid \varepsilon
\end{aligned}$$

Proof omitted.

**2.41** This proof is similar to the proof of the pumping lemma for CFLs. Let $G = (V, \Sigma, S, R)$ be a CFG generating $A$ and let $b$ be the length of the longest right-hand side of a rule in $G$. If a variable $A$ in $G$ generates a string $s$ with a parse tree of height at most $h$, then $|s| \leq b^h$.

Let $p = b^{|V|+2}$ where $|V|$ is the number of variables in $G$. Let $s_1 = \mathtt{a}^{2p!}\mathtt{b}^{p!}\mathtt{c}^{p!}$. Consider a parse tree for $s_1$ that has the fewest possible nodes. Take a symbol c in $s_1$ which has the longest path to $S$ in the parse tree among all c's. That path must be longer

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



than $|V|+1$ so some variable $B$ must repeat on this path. Chose a repetition that occurs among the lowest (nearest to the leaves) $|V|+1$ variables. The upper $B$ can have at most $b^{|V|+2} \leq p$ c's in the subtree below it and therefore it can have at most that number of b's or else the tree surgery technique would yield a derived string with unequal numbers of b's and c's. Hence the subtree below the upper $B$ doesn't produce any a's. Thus the upper and lower $B$'s yield a division of $s_1$ into five parts $uvxyz$ where $v$ contains only b's and $y$ contains only c's. Any other division would allow us to pump the result to obtain a string outside of $A$. Let $q = |v| = |y|$. Because $q \leq p$ we know that $q$ divides $p!$. So we can pump this string up and get a parse tree for $\mathtt{a}^{2p!}\mathtt{b}^{2p!}\mathtt{c}^{2p!}$.

Next let $s_2 = \mathtt{a}^{p!}\mathtt{b}^{p!}\mathtt{c}^{2p!}$ and carry out the same procedure to get a different parse tree for $\mathtt{a}^{2p!}\mathtt{b}^{2p!}\mathtt{c}^{2p!}$. Thus this string has two different parse trees and therefore $G$ is ambiguous.

**2.42** **a.** Assume $A$ is context-free. Let $p$ be the pumping length given by the pumping lemma. We show that $s = \mathtt{0}^p\mathtt{1}^p\mathtt{0}^p\mathtt{1}^p$ cannot be pumped. Let $s = uvxyz$.
If either $v$ or $y$ contain more than one type of alphabet symbol, $uv^2xy^2z$ does not contain the symbols in the correct order and cannot be a member of $A$. If both $v$ and $y$ contain (at most) one type of alphabet symbol, $uv^2xy^2z$ contains runs of 0's and 1's of unequal length and cannot be a member of $A$. Because $s$ cannot be pumped without violating the pumping lemma conditions, $A$ is not context free.

**d.** Assume $A$ is context-free. Let $p$ be the pumping length from the pumping lemma. Let $s = \mathtt{a}^p\mathtt{b}^p\#\mathtt{a}^p\mathtt{b}^p$. We show that $s = uvxyz$ cannot be pumped. Use the same reasoning as in part (c).

**2.43** Assume $B$ is context-free and get its pumping length $p$ from the pumping lemma. Let $s = \mathtt{0}^p\mathtt{1}^{2p}\mathtt{0}^p$. Because $s \in B$, it can be split $s = uvxyz$ satisfying the conditions of the lemma. We consider several cases.

i) If both $v$ and $y$ contain only 0's (or only 1's), then $uv^2xy^2z$ has unequal numbers of 0s and 1s and hence won't be in $B$.

ii) If $v$ contains only 0s and $y$ contains only 1s, or vice versa, then $uv^2xy^2z$ isn't a palindrome and hence won't be in $B$.

iii) If both $v$ and $y$ contain both 0s and 1s, condition 3 is violated so this case cannot occur.

iv) If one of $v$ and $y$ contain both 0s and 1s, then $uv^2xy^2z$ isn't a palindrome and hence won't be in $B$.

Hence $s$ cannot be pumped and contradiction is reached. Therefore $B$ isn't context-free.

**2.44** Assume $C$ is context-free and get its pumping length $p$ from the pumping lemma. Let $s = \mathtt{1}^p\mathtt{3}^p\mathtt{2}^p\mathtt{4}^p$. Because $s \in C$, it can be split $s = uvxyz$ satisfying the conditions of the lemma. By condition 3, $vxy$ cannot contain both 1s and 2s, and cannot contain both 3s and 4s. Hence $uv^2xy^2z$ doesn't have equal number of 1s and 2s or of 3s and 4s, and therefore won't be a member of $C$, so $s$ cannot be pumped and contradiction is reached. Therefore $C$ isn't context-free.

**2.45** Assume to the contrary that $F$ is a CFL. Let $p$ be the pumping length given by the pumping lemma. Let $s = \mathtt{a}^{2p^2}\mathtt{b}^{2p}$ be a string in $F$. The pumping lemma says that we can divide $s = uvxyz$ satisfying the three conditions. We consider several cases. Recall that condition three says that $|vxy| \leq p$.

i) If either $v$ or $y$ contain two types of symbols, $uv^2xy^2z$ contains some b's before a's and is not in $F$.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



ii) If both $v$ and $y$ contain only a's, $uv^2xy^2z$ has the form $\mathtt{a}^{2p^2+l}\mathtt{b}^{2p}$ where $l \leq p < 2p$. But $2p^2 + l$ isn't a multiple of $2p$ if $l < 2p$ so $uv^2xy^2z \notin F$.

iii) If both $v$ and $y$ contain only b's, $uv^mxy^mz$ for $m = 2p^2$ has the form $\mathtt{a}^i\mathtt{b}^j$ where $i < j$ and so it cannot be in $F$.

iv) If $v$ contains only a's and $y$ contains only b's, let $t = uv^mxy^mz$. String $t$ cannot be member of $F$ for any sufficiently large value of $m$. Write $s$ as $\mathtt{a}^{g+|v|}\mathtt{b}^{h+|y|}$. Then $t$ is $\mathtt{a}^{g+m|v|}\mathtt{b}^{h+m|y|}$. If $t \in F$ then $g + m|v| = k(h + m|y|)$ for some integer $k$. In other words

$$k = \frac{g + m|v|}{h + m|y|}.$$

If $m$ is sufficiently large, $g$ is a tiny fraction of $m|v|$ and $h$ is a tiny fraction of $m|y|$ so we have

$$k = \frac{m|v|}{m|y|} = \frac{|v|}{|y|}$$

because $k$ is an integer. Moreover $k < p$. Rewriting the first displayed equation we have $g - hk = m(k|y| - |v|)$ which must be 0, due to the second displayed equation. But we have chosen $s$ so that $g - hk \neq 0$ for $k < p$, so $t$ cannot be member of $F$.

Thus none of the cases can occur, so the string $s$ cannot be pumped, a contradiction.

**2.46** $L(G)$ is the language of strings of 0s and #s that either contain exactly two #s and any number of 0s, or contain exactly one # and the number of 0s to the right of the # is twice the number of 0s to the left. First we show that three is not a pumping length for this language. The string 0#00 has length at least three, and it is in $L(G)$. It cannot be pumped using $p = 3$, because the only way to divide it into $uvxyz$ satisfying the first two conditions of the pumping lemma is $u = z = \varepsilon$, $v = \mathtt{0}$, $x = \mathtt{\#}$, and $y = \mathtt{00}$, but that division fails to satisfy the third condition.

Next, we show that 4 is a pumping length for $L(G)$. If $w \in L(G)$ has length at least 4 and if it contains two #s, then it contains at least one 0. Therefore, by cutting $w$ into $uvxyz$ where either $v$ or $y$ is the string 0, we obtain a way to pump $w$. If $w \in L(G)$ has length at least 4 and if it contains a single #, then it must be of the form $\mathtt{0}^k\mathtt{\#}\mathtt{0}^{2k}$ for some $k \geq 1$. Hence, by assigning $u = \mathtt{0}^{k-1}$, $v = \mathtt{0}$, $x = \mathtt{\#}$, $y = \mathtt{00}$, and $z = \mathtt{0}^{2k-2}$, we satisfy all three conditions of the lemma.

**2.47** Assume G generates a string $w$ using a derivation with at least $2^b$ steps. Let $n$ be the length of $w$. By the results of Problem 2.38, $n \geq \frac{2^b+1}{2} > 2^{b-1}$.

Consider a parse tree of $w$. The right-hand side of each rule contains at most two variables, so each node of the parse tree has at most two children. Additionally, the length of $w$ is at least $2^b$, so the parse tree of $w$ must have height at least $b+1$ to generate a string of length at least $2^b$. Hence, the tree contains a path with at least $b+1$ variables, and therefore some variable is repeated on that path. Using a surgery on trees argument identical to the one used in the proof of the CFL pumping lemma, we can now divide w into pieces $uvxyz$ where $uv^ixy^iz \in G$ for all $i \geq 0$. Therefore, $L(G)$ is infinite.

**2.48** Let $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^l \mid i, j, k, l \geq 0 \text{ and if } i = 1 \text{ then } j = k = l\}$. $F$ is not context free because $F \cap \mathtt{ab}^*\mathtt{c}^*\mathtt{d}^* = \{\mathtt{ab}^n\mathtt{c}^n\mathtt{d}^n \mid n \geq 0\}$, which is not a CFL by the pumping lemma, and because the intersection of a CFL and a regular language is a CFL. However, $F$ does satisfy the pumping lemma with $p = 2$. ($p = 1$ works, too, with a bit more effort.)

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



If $s \in F$ has length 2 or more then:

i) If $s \in \mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ write $s$ as $s = rgt$ for $g \in \{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and divide $s = uvxyz$ where $u = r$, $v = g$, $x = y = \varepsilon$, $z = t$.

ii) If $s \in \mathtt{ab}^*\mathtt{c}^*\mathtt{d}^*$ write $s$ as $s = \mathtt{a}t$ and divide $s = uvxyz$ where $u = \varepsilon$, $v = \mathtt{a}$, $x = y = \varepsilon$, $z = t$.

iii) If $s \in \mathtt{aaa}^*\mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ write $s$ as $s = \mathtt{aa}t$ and divide $s = uvxyz$ where $u = \varepsilon$, $v = \mathtt{aa}$, $x = y = \varepsilon$, $z = t$.

In each of the three cases, we may easily check that the division of $s$ satisfies the conditions of the pumping lemma.

**2.49** Let $G = (\Sigma, V, T, P, S)$ be a context free grammar for $A$. Define $r$ to be the length of the longest string of symbols occurring on the right hand side of any production in $G$. We set the pumping length $k$ to be $r^{2|V|+1}$. Let $s$ be any string in $A$ of length at least $k$ and let $T$ be a derivation tree for $A$ with the fewest number of nodes. Observe that since $s \geq k$, the depth of $T$ must be at least $2|V| + 1$. Thus, some path $p$ in $T$ has length at least $2|V| + 1$. By the pigeonhole principle, some variable $V'$ appears at least three times in $p$. Label the last three occurrences of $V'$ in $p$ as $V_1$, $V_2$, and $V_3$. Moreover, define the strings generated by $V_1$, $V_2$, and $V_3$ in $T$ as $s_1$, $s_2$, and $s_3$ respectively. Now, each of these strings is nested in the previous because they are all being generated on the same path. Suppose then that $s_1 = l_1 s_2 r_1$ and $s_2 = l_2 s_3 r_2$. We now have three cases to consider:

i) $|l_1 l_2| \geq 1$ and $|r_1 r_2| \geq 1$: In this case, there is nothing else to prove since we can simply pump in the usual way.

ii) $|l_1 l_2| = 0$: Since we defined $T$ to be a minimal tree, neither $r_1$ nor $r_2$ can be the empty string. So, we can now pump $r_1$ and $r_2$.

iii) $|r_1 r_2| = 0$: This case is handled like the previous one.

**2.51** Let $A$ and $B$ be defined as in the solution to Problem 2.50. Let $D$ be the shuffle of $A$ and $B$. Then let $E = D \cap ((\mathtt{0} \cup \mathtt{1})(\mathtt{a} \cup \mathtt{b}))^*$ The language $E$ is identical to the language $C$ in Problem 2.50 and was shown there to be non-context free. The intersection of a CFL and a regular language is a CFL, so $D$ must not be context free, and therefore the class of CFLs isn't closed under the shuffle operation.

**2.52** The language $C$ is context free, therefore the pumping lemma holds. Let $p$ be the pumping length and $s \in C$ be a string longer then $p$, so it can be split in $uvxyz$ such that $uv^ixy^iz \in C$ for all $i > 0$, where $|vy| > 0$. All prefixes are in $C$ so all $uv^i \in C$ and thus the regular language $u(v)^* \subseteq C$. If $v \neq \varepsilon$, that language is infinite and we're done. If $v = \varepsilon$, then $y \neq \varepsilon$, and the infinite regular language $uvx(y)^* \subseteq C$.

**2.53** **a.** Let $B = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \neq j \text{ or } i \leq k\}$ which is a CFL. Let $s = \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \in B$. Assume $k \geq 2$. Let $s' = \mathtt{a}^i\mathtt{b}^j\mathtt{c}^{k-1}$ which is a prefix of $s$. Show that $s' \in B$ if $i \neq j$ or $i \neq k$.

**1.** If $i \neq j$ then $s' \in B$.

**2.** If $i < k$ then $i \leq (k-1)$ so $s' \in B$.

**3.** If $i > k$ then $i \neq j$ (because $s \in B$) so $s' \in B$.

If $i = j = k$ then $s$ has no prefix in $B$ because removing some of the c's would yield a string that fails both conditions of membership in $B$. Thus, $(\textit{NOPREFIX}(B) \cap \mathtt{a}^*\mathtt{b}^*\mathtt{ccc}^*) \cup \mathtt{abc} \cup \varepsilon = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \geq 0\}$ which isn't a CFL. Therefore, $\textit{NOPREFIX}(B)$ cannot be a CFL.

**b.** Let $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \neq j \text{ or } i \geq k\}$ which is a CFL. Let $s = \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \in C$. If $i \neq j$ or if $i > k$ then $s\mathtt{c} \in C$. But if $i = j = k$ then $s$ has no extension in $C$. Therefore, $\textit{NOEXTEND}(C) = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \geq 0\}$ which isn't a CFL.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



**2.54** Assume $Y$ is a CFL and let $p$ be the pumping length given by the pumping lemma. Let $s = 1^{p+1}\#1^{p+2}\#\cdots\#1^{5p}$. String $s$ is in $Y$ but we show it cannot be pumped. Let $s = uvxyz$ satisfying the three conditions of the lemma. Consider several cases.

i) If either $v$ or $y$ contain #, the string $uv^3xy^3z$ has two consecutive $t_i$'s which are equal to each other. Hence that string is not a member of $Y$.

ii) If both $v$ and $y$ contain only 1s, these strings must either lie in the same run of 1s or in consecutive runs of 1s within $s$, by virtue of condition 3. If $v$ lies within the runs from $1^{p+1}$ to $1^{3p}$ then $uv^2xy^2z$ adds at most $p$ 1s to that run so that it will contain the same number of 1s in a higher run. Therefore the resulting string will not be a member of $Y$. If $v$ lies within the runs from $1^{3p+1}$ to $1^{5p}$ then $uv^0xy^0z$ subtracts at most $p$ 1s to that run so that it will contain the same number of 1s in a lower run. Therefore the resulting string will not be a member of $Y$.

The string $s$ isn't pumpable and therefore doesn't satisfy the conditions of the pumping lemma, so a contradiction has occurred. Hence $Y$ is not a CFL.

**2.55** **a.** First note that a PDA can use its stack to simulate an unbounded integer counter. Next suppose $A \subseteq \{0,1\}^*$ is recognized by a DFA $D$. Clearly, a binary string $x$ is in *SCRAMBLE*$(A)$ iff $x$ has the same length and same number of 1s as some $w$ that is accepted by $D$. Thus, a PDA $M$ for *SCRAMBLE*$(A)$ operates on input $x$ by nondeterministically simulating $D$ on every possible $|x|$-long input $w$ using its stack as a counter to keep track of the difference between the number of 1s in $w$ and the number of 1s in $x$. A branch of $M$'s computation accepts iff it reaches the end of $x$ with the stack recording a difference of 0 and the simulation of $D$ on an accepting state.

**b.** If $|\Sigma| > 2$, then the claim in part (a) is false. For example, consider the regular language $A = (\mathtt{abc})^*$ over the ternary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

$$\textit{SCRAMBLE}(A) = \{x \in \Sigma^* \mid x \text{ has the same number of a's, b's and c's.}\}$$

This language is not context-free.

**2.56** Let $M_A$ be a DFA that recognizes $A$, and $M_B$ be a DFA that recognizes $B$. We construct a PDA recognizing $A \diamond B$. This PDA simulates $M_A$ on the first part of the string pushing every symbol it reads on the stack until it guesses that it has reached the middle of the input. After that it simulates $M_B$ on the remaining part of the string popping the stack for every symbol it reads. If the stack is empty at the end of the input and both $M_A$ and $M_B$ accepted, the PDA accepts. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, the PDA rejects on that branch of the computation.

**2.57** Suppose $A$ is context-free and let $p$ be the associated pumping length. Let $s = 1^{2p}0^p1^p1^{2p}$ which is in $A$ and longer than $p$. By the pumping lemma we know that $s = uvxyz$ satisfying the three conditions. We distinguish cases to show that $s$ cannot be pumped and remain in $A$.

- If $vxy$ is entirely in the last two thirds of $s$, then $uv^2xy^2z$ contains 0s in its first third but not in its last third and so is not in $A$.
- Otherwise, $vxy$ intersects the first third of $s$, and it cannot extend beyond the first half of $s$ without violating the third condition.
  - If $v$ contains both 1s and 0s, then $uv^2xy^2z$ contains 0s in its first third but not in its last third and so is not in $A$.
  - If $v$ is empty and $y$ contains both 0s and possibly 1s, then again $uv^2xy^2z$ contains 0s in its first third but not in its last third and is not in $A$.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

CENGAGE Learning



– Otherwise, either $v$ contains only 1s, or $v$ is empty and $y$ contains only 1s. In both cases, $uv^{1+6p}xy^{1+6p}z$ contains 0s in its last third but not in its first third and so is not in $A$.

A contradiction therefore arises and so $A$ isn't a CFL.

**2.58** **a.**

$$\begin{aligned} S &\to XSX \mid T1 \\ T &\to XT \mid X \\ X &\to 0 \mid 1 \end{aligned}$$

Here $T$ is any nonempty string, and $S$ is any string with $T1$ in the middle (so the 1 falls at least one character to the right of the middle).

**b.** Read the input until a 1 appears, while at the same time pushing 0s. Pop two 0s. Continue reading the input while popping the stack in an accept state until reach the end of the input. If the stack empties before reaching the end of the input, then do not read any more input (i.e., reject).

**2.59** **a.**

$$\begin{aligned} S &\to T \mid 1 \\ T &\to XXTX \mid XTXX \mid XTX \mid X1X \\ X &\to 0 \mid 1 \end{aligned}$$

**b.** Assume that $C_2$ is a CFG and apply the pumping lemma to obtain the pumping length $p$. Let $s = 0^{p+2}10^p10^{p+2}$. Clearly, $s \in C_2$ so we may write $s = uvxyz$ satisfying the three conditions of the pumping lemma. If either $v$ or $y$ contains a 1, then the string $uxz$ contains fewer than two 1s and thus it cannot be a member of $C_2$. By condition three of the pumping lemma, parts $v$ and $y$ cannot together contain 0s from both of the two outer runs of 0s. Hence in the string $uv^2xy^2z$ the 1s cannot both remain in the middle third and so $uv^2xy^2z$ is not in $C_2$.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

© 2013 Cengage Learning. All Rights Reserved. This edition is intended for use outside of the U.S. only, with content that may be different from the U.S. Edition. May not be scanned, copied, duplicated, or posted to a publicly accessible website, in whole or in part.

# Introduction to the Theory of Computation Solutions

Ryan Dougherty

# Contents

**1 Solutions** **5**
1.1 Chapter 1 . . . . . 6
1.2 Chapter 2 . . . . . 12
1.3 Chapter 3 . . . . . 14
1.4 Chapter 4 . . . . . 17
1.5 Chapter 5 . . . . . 20
1.6 Chapter 6 . . . . . 23
1.7 Chapter 7 . . . . . 25
1.8 Chapter 8 . . . . . 28
1.9 Chapter 9 . . . . . 29
1.10 Chapter 10 . . . . . 30

# Chapter 1

# Solutions

## 1.1 Chapter 1

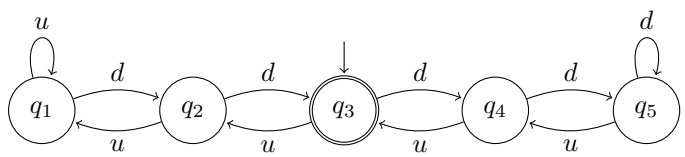
1.1 The following are the state diagrams of two DFAs, $M_1$ and $M_2$. Answer the following questions about each of these machines.
**Solution:** answered in the text.

1.2 Give the formal description of the machines $M_1$ and $M_2$ pictured in Exercise 1.1.
**Solution:** answered in the text.

1.3 The formal description of a DFA $M$ is $(\{q_1, q_2, q_3, q_4, q_5\}, \{u, d\}, \delta, q_3, \{q_3\})$, where $\delta$ is given by the following table. Give the state diagram of this machine.

| | $u$ | $d$ |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_1$ | $q_3$ |
| $q_3$ | $q_2$ | $q_4$ |
| $q_4$ | $q_3$ | $q_5$ |
| $q_5$ | $q_4$ | $q_5$ |

**Solution:**




1.4 Each of the following languages is the intersection of two simpler languages. In each part, construct DFAs for the simpler languages, then combine them using the construction discussed in footnote 3 (page 46) to give the state diagram of a DFA for the language given. In all parts, $\Sigma = \{a, b\}$.

**Note:** for simplicity, I omit the state diagrams.

a. **Solution:** The state set is $Q = \{q_{i,j} : 0 \le i \le 3, 0 \le j \le 2\}$, with transition function:
   - $\delta(q_{i,j}, a) = q_{i+1,j}$ for all $0 \le i \le 2, 0 \le j \le 2$,
   - $\delta(q_{i,j}, b) = q_{i,j+1}$ for all $0 \le i \le 3, 0 \le j \le 1$.

   The start state is $q_{0,0}$, and the final state is $q_{3,2}$.

b. **Solution:** answered in the text.

c. **Solution:** the state set is $Q = \{q_{i,j} : i \in \{0, 1, 2, 3\}, j \in \{even, odd\}\}$, with transition function:
   - $\delta(q_{i,even}, a) = q_{i,odd}$ for all $i$,
   - $\delta(q_{i,odd}, a) = q_{i,even}$ for all $i$,
   - $\delta(q_{i,j}, b) = q_{i+1,j}$ for $i \in \{0, 1, 2\}, j \in \{even, odd\}$,
   - $\delta(q_{3,j}, b) = q_{3,j}$ for $j \in \{even, odd\}$.

d. **Solution:** answered in the text.

e. **Solution:** the state set is $Q = \{ij : i, j \in \{0, 1, 2\}\}$, with transition function in Table 1.1.

f. **Solution:** the state set is $Q = \{ij : i, j \in \{0, 1\}\}$, with transition function in Table 1.2.

g. **Solution:** the state set is $Q = \{ij : i, j \in \{0, 1\}\}$, with transition function in Table 1.3.

1.5 Each of the following languages is the complement of a simpler languages. In each part, construct DFAs for the simpler language, then use it to give the state diagram of a DFA for the language given. In all parts, $\Sigma = \{a, b\}$.

**Note:** for simplicity, I omit the state diagrams.

a. **Solution:** answered in the text.

| State ↓ Symbol → | $a$ | $b$ |
|---|---|---|
| 00 (start) | 10 | 21 |
| 01 | 11 | 22 |
| 02 | 12 | 22 |
| 10 (final) | 10 | 11 |
| 11 (final) | 11 | 12 |
| 12 | 12 | 12 |
| 20 | 20 | 21 |
| 21 | 21 | 22 |
| 22 | 22 | 22 |

Table 1.1: 1.4e state table.

| State ↓ Symbol → | $a$ | $b$ |
|---|---|---|
| 00 (start) | 01 | 10 |
| 01 | 00 | 11 |
| 10 | 01 | 00 |
| 11 (final) | 00 | 01 |

Table 1.2: 1.4f state table.

| State ↓ Symbol → | $a$ | $b$ |
|---|---|---|
| 00 (start) | 11 | 10 |
| 01 (final) | 10 | 11 |
| 10 | 01 | 00 |
| 11 | 00 | 01 |

Table 1.3: 1.4g state table.

b. **Solution:** answered in the text.

1.6 Give state diagrams of NFAs with the specified number of states recognizing each of the following languages. In all parts, the alphabet is {0, 1}.

a. **Solution:** answered in the text.

f. **Solution:** answered in the text.

1.7 Each of the following languages is the complement of a simpler language. In each part, construct a DFA for the simpler language, then use it to give the state diagram of a DFA for the language given. In all parts, $\Sigma = \{a, b\}$.

a. **Solution:** answered in the text.

b. **Solution:** answered in the text.

1.11 Prove that every NFA can be converted to an equivalent one that has a single accept state.
**Solution:** answered in the text.

1.20 For each of the following languages, give two strings that are members and two strings that are not members–a total of four strings for each part. Assume the alphabet $\Sigma = \{a, b\}$ in all parts.

a. **Solution:** 2 members: $ab, aa$; 2 not members: $ba, bab$.

b. **Solution:** 2 members: $ab, abab$; 2 not members: $b, ba$.

c. **Solution:** 2 members: $aa, bb$; 2 not members: $ba, ab$.

d. **Solution:** 2 members: $\epsilon, aaa$; 2 not members: $b, bb$.

e. **Solution:** 2 members: $aba, aaba$; 2 not members: $b, bb$.

f. **Solution:** 2 members: $aba, bab$; 2 not members: $a, b$.

g. **Solution:** 2 members: $b, ab$; 2 not members: $ba, bb$.

h. **Solution:** 2 members: $a, ba$; 2 not members: $ab, b$.

1.23 Let $B$ be any language over the alphabet $\Sigma$. Prove that $B = B^+$ iff $BB \subseteq B$.
**Solution:** answered in the text.

1.24 A **finite state transducer (FST)** is a type of deterministic finite automaton whose output is a string and not just *accept* or *reject*. The state diagrams for finite state transducers $T_1$ and $T_2$ are shown in the text.
Give the sequence of states entered and the output produced in each of the following parts.

a. $T_1$ on input *011*
**Solution:** 000

b. $T_1$ on input *211*
**Solution:** 111

c. $T_1$ on input *121*
**Solution:** 011

d. $T_1$ on input *0202*
**Solution:** 0101

e. $T_2$ on input $b$
**Solution:** 1

f. $T_2$ on input $bbab$
**Solution:** 1111

g. $T_2$ on input $bbbbbb$
**Solution:** 110110

h. $T_2$ on input $\epsilon$
**Solution:** $\epsilon$

1.29 Use the pumping lemma to show that the following languages are not regular.

a. **Solution:** answered in the text.

b. **Solution:** $A_2 = \{www \mid w \in \{a,b\}^*\}$. Suppose that $A_2$ were regular, and let $p$ be the constant for $A_2$ as given by the pumping lemma. Then, let $s = a^p b a^p b a^p b \in A_2$. Therefore, we can decompose $s$ as $xyz$, where $|xy| \leq p$, $|y| > 0$, and $xy^iz \in A_2$ for $\forall i \in \mathbb{N}$. Since $|xy| \leq p$, $x, y$ consist entirely of $a$'s; therefore, we can write $x = a^c$, $y = a^d$, and $z = a^{p-c-d}ba^pba^pb$, where $c \geq 0, d > 0, p-c-d \geq 0$. For the third condition, choose $i = 2$: $w = xy^2z = a^ca^{2d}a^{p-c-d}ba^pba^pb = a^{p+d}ba^pba^pb$. Since $d > 0$, $w \notin A_2$: therefore, we have a contradiction, and $A_2$ is not regular.

c. **Solution:** answered in the text.

1.31 For any string $w = w_1w_2 \cdots w_n$, the ***reverse*** of $w$, written $w^{\mathcal{R}}$, is the string $w$ in reverse order, $w_n \cdots w_2w_1$. For any language $A$, let $A^{\mathcal{R}} = \{w^{\mathcal{R}} | w \in A\}$. Show that if $A$ is regular, so is $A^{\mathcal{R}}$.
**Solution:** For any regular language $A$, let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA recognizing $A$. We need to construct an NFA/DFA $N$ such that $L(N) = A^{\mathcal{R}}$. Let $N = (Q', \Sigma, \delta', q_0', \{q_0\})$, where $q_0' \notin Q$ and $Q' = Q \cup \{q_0'\}$. Define $\delta'$ as: $\delta'(q_0', \epsilon) = F$, and $\delta'(q_0', a) = \emptyset, \forall a \in \Sigma$. Also, $\forall (q, a) \in Q \times \Sigma, \delta'(q, a) = \{q' | \delta(q', a) = q\}$.

Another way to approach this problem is an informal explanation: we reverse all of the transitions of $M$, and set the accept state of $N$ to be $M$'s start state. Also, introduce a new state $q_0'$ as $N$'s start state, which goes to every accept state in $M$ by an $\epsilon$-transition.

1.39 The construction in Theorem 1.54 shows that every GNFA is equivalent to a GNFA with only two states. We can show that an opposite phenomenon occurs for DFAs. Prove that for every $k > 1$, a language $A_k \subseteq \{0,1\}^*$ exists that is recognized by a DFA with $k$ states but not by one with only $k-1$ states.
**Solution:** let $L_k = \{0^i \mid i \geq k-1\}$. This language consists of strings of length $\geq k-1$. Therefore, no DFA of $k-1$ states can recognize this language, but certainly one of $k$ states can.

1.40 Recall that string $x$ is a **prefix** of string $y$ if a string $z$ exists where $xz = y$, and that $x$ is a **proper prefix** of $y$ if in addition $x \neq y$. In each of the following parts, we define an operation on a language $A$. Show that the class of regular languages is closed under that operation.

a. **Solution:** answered in the text.

1.42 For languages $A$ and $B$, let the ***shuffle*** of $A$ and $B$ be the language $\{w \mid w = a_1b_1...a_kb_k$, where $a_1...a_k \in A$ and $b_1...b_k \in B$, each $a_i, b_i \in \Sigma^*\}$. Show that the class of regular languages is closed under shuffle.
**Solution:** Let $D_A = (Q_A, \Sigma, \delta_A, q_A, F_A$ and $D_B = (Q_B, \Sigma, \delta_B, q_B, F_B$ be the DFAs recognize $A$ and $B$, respectively. We will design a DFA $D = (Q, \Sigma, \delta, q_0, F)$ such that it recognizes the shuffle of $A$ and $B$ as follows:

- $Q = Q_A \times Q_B \times \{A, B\}$.

- $q_0 = (q_A, q_B, A)$.
- $F = F_A \times F_B \times \{A\}$.
- For $\delta$:
  - $\delta((x, y, A), a) = (\delta_A(x, a), y, B)$.
  - $\delta((x, y, B), b) = (x, \delta_B(y, b), A)$.

1.44 Let $B$ and $C$ be languages over $\Sigma = \{0, 1\}$. Define

$$B \xleftarrow{1} C = \{w \in B | \text{ for some } y \in C, \text{ strings } w \text{ and } y \text{ contain equal numbers of 1s}\}.$$

Show that the class of regular languages is closed under the $\xleftarrow{1}$ operation.
**Solution:** answered in the text.

1.45 Let $A/B = \{w | wx \in A \text{ for some } x \in B\}$. Show that if $A$ is regular and $B$ is any language, then $A/B$ is regular.
**Solution:** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA such that $L(M) = A$, and $\Sigma$ is the union of the alphabets of $A$ and $B$. Let $F_r = \{q \in Q | \exists x \in B$ such that $M$ can reach a final state when having read $x$, starting at $q\}$. Therefore, $L(M) = A/B$.

1.48 Let $\Sigma = \{0, 1\}$ and let

$$D = \{w | w \text{ contains an equal number of occurrences of the substrings 01 and 10}\}.$$

Thus $101 \in D$ because 101 contains a single 01 and a single 10, but $1010 \notin D$ because 1010 contains two 10s and one 01. Show that $D$ is a regular language.
**Solution:** $D$ is just precisely described by the regular expression $(1^+0^*1^+)^* \cup (0^+1^*0^+)^*$

1.50 Read the informal definition of the finite state transducer given in Exercise 1.24. Prove that no FST can output $w^R$ for every input $w$ if the input and output alphabets are $\{0, 1\}$.
**Solution:** answered in the text.

1.52 **Myhill-Nerode theorem**. Refer to Problem 1.51. Let $L$ be a language and let $X$ be a set of strings. Say that $X$ is ***pairwise distinguishable by L*** if every two distinct strings in $X$ are distinguishable by $L$. Define the ***index of L*** to be the maximum number of elements in any set that is pairwise distinguishable by $L$. The index of $L$ may be finite or infinite.
**Solution:** answered in the text.

1.53 Let $\Sigma = \{0, 1, +, =\}$ and $ADD = \{\ x = y + z \mid x, y, z$ are binary integers, and $x$ is the sum of $y$ and $z\}$. Show that $ADD$ is not regular.
**Solution:** Assume that $ADD$ is regular. Therefore, there exists a pumping length $p \in \mathbb{Z}$ such that the three conditions of the Pumping Lemma are satisfied. Choose $w$ as $1^p = 0^p + 1^p$. Clearly, $w \in ADD$. By the conditions of the Pumping Lemma, we can partition $w = xyz$ such that $|xy| \leq p, |y| > 0$, and $xy^iz \in ADD$ for $\forall i \in \mathbb{N}$. By the first and second conditions of the Pumping Lemma, $x$ and $y$ consist entirely of 1s, i.e. $x = 1^a, y = 1^b, z = 1^{p-a-b} = 0^p + 1^p$ for $a \geq 0, b > 0$. By the third condition, $xy^iz \in ADD$ for $\forall i \in \mathbb{N}$. Choose $i = 2$: $xy^2z = 1^a1^{2b}1^{p-a-b} = 0^p + 1^p = 1^{p+b} = 0^p + 1^p$. However, this string is not in $ADD$, because this implies $p + b = p$, but $b > 0$, a contradiction. Therefore, $ADD$ is not regular.

1.55 The pumping lemma says that every regular language has a pumping length $p$, such that every string in the language can be pumped if it has length $p$ or more. If $p$ is a pumping length for language $A$, so is any length $p' \geq p$. The minimum pumping length for $A$ is the smallest $p$ that is a pumping length for $A$. For example, if $A = 01^*$, the minimum pumping length is 2. The reason is that the string $s = 0$ is in $A$ and has length 1 yet $s$ cannot be pumped; but any string in $A$ of length 2 or more contains a 1 and hence can be pumped by dividing it so that $x = 0, y = 1$, and $z$ is the rest. For each of the following languages, give the minimum pumping length and justify your answer.

a. $0001^*$
**Solution:** answered in the text.

b. $0^*1^*$
**Solution:** answered in the text.

c. $001 \cup 0^*1^*$
**Solution:** We cannot pump 001, so the minimum pumping length is 4.

d. $0^*1^+0^+1^* \cup 10^*1$
**Solution:** answered in the text.

e. $(01)^*$
**Solution:** We cannot pump $\epsilon$, so the minimum pumping length is 1 (if we wanted to be constructive, the answer would be 2, since there is no string of length 1 here).

g. $1^*01^*01^*$
**Solution:** We cannot pump 00, but we can for 100, so the minimum pumping length is 3.

1.58 If $A$ is any language, let $A_{\frac{1}{3}-\frac{1}{3}}$ be the set of all strings in $A$ with their middle thirds removed so that

$$A_{\frac{1}{3}-\frac{1}{3}} = \{xz \mid \text{for some } y,\ |x| = |y| = |z| \text{ and } xyz \in A\}.$$

Show that if $A$ is regular, then $A_{\frac{1}{3}-\frac{1}{3}}$ is not necessarily regular.
**Solution:** Let $A = \{0^*\#1^*\}$, which is regular. Therefore, $A_{\frac{1}{3}-\frac{1}{3}} \cap \{0^*1^*\} = \{0^n1^n \mid n \geq 0\}$ is also regular since regular languages are closed under intersection, and $\{0^*1^*\}$ is a regular language. However, the resulting language is not regular, so therefore $A_{\frac{1}{3}-\frac{1}{3}}$ is not necessarily regular when $A$ is.

1.63 (a) Let $A$ be an infinite regular language. Prove that $A$ can be split into two infinite disjoint regular subsets.
**Solution:** since $A$ is infinite and regular, then the conditions of the Pumping Lemma for Regular Languages hold, i.e., that there is a $p \geq 0$ such that for all $w \in A$, we can partition $w$ into $w = xyz$ that satisfy those 3 conditions. Consider $A' = \{xy^{2i}z \mid i \geq 0\}$, and $A'' = L \cap \overline{A'}$. These are the desired languages because they partition $A$ and are infinite.

(b) Let $B$ and $D$ be two languages. Write $B \Subset D$ if $B \subseteq D$ and $D$ contains infinitely many strings that are not in $B$. Show that if $B$ and $D$ are two regular languages where $B \Subset D$, then we can find a regular language $C$ where $B \Subset C \Subset D$.
**Solution:** Let $L = D \cap \overline{B}$; $L$ is regular by the closure operations, and is infinite. By the Pumping Lemma for Regular Languages, there is a $w \in L$ such that $w = xyz$ such that $|y| > 0$, and for all $i \geq 0$, $xy^iz \in L$. Let $C = B \cup \{xy^iz \in L \mid i \text{ is even}\}$. Call the second language $L'$. We can see that $C$ is regular. Since $L' \cap \overline{B}$ is infinite, we have $B \Subset C$. By a similar analysis, we have $C \Subset D$.

1.70 We define the ***avoids*** operation for languages $A$ and $B$ to be

$$A \textit{ avoids } B = \{w \mid w \in A \text{ and } w \text{ doesn't contain any string in } B \text{ as a substring}\}.$$

Prove that the class of regular languages is closed under the *avoids* operation.
**Solution:** The idea is to find a regular language that has strings in $B$ as a substring, and remove from $A$ this language. Therefore, define $L_{substr} = \Sigma^*B\Sigma^*$. Clearly, $L_{substr}$ is regular because it is the concatenation of 2 regular languages. Since regular languages are closed under complement and intersection, then $A \setminus L_{substr} = A \cap \overline{L_{substr}}$ is also regular. But these are precisely the strings that are in $A$ that are not in $L_{substr}$, which is what we want.

1.72 Let $M_1$ and $M_2$ be DFAs that has $k_1$ and $k_2$ states, respectively, and then let $U = L(M_1) \cup L(M_2)$.

a. Show that if $U \neq \emptyset$, then $U$ contains some string $s$, where $|s| < \max(k_1, k_2)$.
**Solution:** Consider a DFA $D = (Q, \Sigma, \delta, q_0, F)$ such that $L(D) \neq \emptyset$. Therefore, if the final state is reachable, transitioning from $D$'s start state to a final state requires at most $|Q| - |F|$ transitions. Since $U \neq \emptyset$, then either $L(M_1) \neq \emptyset$ or $L(M_2) \neq \emptyset$ (or both). We have the following cases:

1. $L(M_1) = \emptyset, L(M_2) \neq \emptyset$. This implies that $M_1$ has no reachable accept states, and $M_2$ has at least one reachable accept state. Also, $U = L(M_2)$. From our observation above, if $M_2$ accepts a string $s$, then $|s| \leq k_2 - |F_2| < k_2 \leq \max(k_1, k_2)$ where $F_2$ is the set of accept states of $M_2$. Therefore, $s \in L(M_2) = U$.
2. $L(M_1) \neq \emptyset, L(M_2) = \emptyset$. This is equivalent to Case 1.
3. $L(M_1), L(M_2) \neq \emptyset$. Therefore, $M_1$ and $M_2$ have at least one reachable accept state. Let $s_1$ be the string of minimal length accepted by $M_1$, and $s_2$ for $M_2$. Let $s \in U$ be such that $|s| = \min(|s_1|, |s_2|)$. We have the following 2 cases:
   3.1. $|s_1| \neq |s_2|$. This implies $|s| = \min(|s_1|, |s_2|) < \max(|s_1|, |s_2|)$. There are two possibilities. If $\max(|s_1|, |s_2|) = |s_1|$, then $|s| = |s_2| < |s_1| < k_1 \leq \max(k_1, k_2)$. Therefore, we are done. The same conclusion is reached if we consider when $\max(|s_1|, |s_2|) = |s_2|$.
   3.2. $|s_1| = |s_2$. This implies $|s| = \min(|s_1|, |s_2|) = \max(|s_1|, |s_2|)$. From our observation above, we have that $|s| = \min(|s_1|, |s_2|) = \max(|s_1|, |s_2|) = |s_1| < k_1 \leq \max(k_1, k_2)$. Therefore, we are done.

b. Show that if $U \neq \Sigma^*$, then $U$ excludes some string $s$, where $|S| < k_1k_2$.

**Solution:** Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA such that $L(D) = U = L(M_1) \cup L(M_2)$. Suppose (to the contrary) that all strings $s \in U$ are such that $|s| < k_1k_2$. Also, suppose there exists a non-accepting state $q \in Q$. Therefore, there cannot exist a sequence of states $q_1, \cdots, q_{k_1k_2}$ in $D$ such that running $D$ on $s$ would have $q$ in this sequence:

1. $q_1 = q_0$.
2. $\delta(q_i, x) = q_{i+1}$ for $1 \leq i \leq k_1k_2 - 1$ and for all $x \in \Sigma$.
3. $q_j \neq q_k$ for $j \neq k$.

If $q$ were in this sequence, then $q$ would be an accepting state. However, this implies that $D$ has $k_1k_2 + 1$ distinct states, and $D$ has only $k_1k_2$ states, a contradiction. Therefore, either $q$ is not reachable from $q_0$, or $q$ does not exist.

Since $q$ is an arbitrary non-accepting state of $D$, the only states reachable from $q_0$ are accepting states. Now, we prove by induction that all strings of length $\geq k_1k_2$ are accepted by $D$.

Basis step: from above, $D$ accepts any string of length $k_1k_2 - 1 < k_1k_2$. Let $s \in \Sigma^*$ such that $|s| = k_1k_2 - 1$. Therefore, $\delta^*(q_0, s)$ must result in an accepting state. Let $r$ be this state. Therefore, $\delta(q, w)$ for all $w \in \Sigma$ must result in an accepting state, since all reachable states from $q_0$ are accepting states. Therefore, $D$ must accept any string $s \in \Sigma^*$ such that $|s| = k_1k_2$.

Inductive Hypothesis: assume that $D$ accepts any string $s \in \Sigma^*$ such that $|s| = k_1k_2 + n$ for some $n \in \mathbb{N}$.

Inductive Step: By the IH, $\delta^*(q_0, s)$ must result in an accepting state for all $s \in \Sigma^*$ such that $|s| = k_1k_2 + n$ for some $n \in \mathbb{N}$. Let the accept state be $r$. For all $w \in \Sigma, \delta(q, w)$ results in an accepting state. It follows that $D$ accepts all $s \in \Sigma^*$ such that $|s| = k_1k_2 + n + 1$. Now, we showed that $D$ accepts all $s \in \Sigma^*$ such that $|s| \geq k_1k_2$. From earlier, we showed that $D$ accepts all $t \in \Sigma^*$ such that $|t| < k_1k_2$.

Therefore, $L(D) = U = \Sigma^*$. However, this is a contradiction to our assumption. Therefore, $U$ excludes some string $s$ where $|s| < k_1k_2$.

## 1.2 Chapter 2

2.3 Answer each part for the following context-free grammar $G$.

$$\begin{aligned} R &\to XRX|S \\ S &\to aTb|bTa \\ T &\to XTX|X|\epsilon \\ X &\to a|b \end{aligned}$$

**Solution:** answered in the text.

2.7 Give informal English descriptions of PDAs for the languages in Exercise 2.6.
**Solution:** answered in the text.

2.8 Show that the string `the girl touches the boy with the flower` has two different leftmost derivations in grammar $G_2$ on page 103. Describe in English the two different meanings of this sentence.
**Solution:** answered in the text.

2.9 Give a context-free grammar that generates the language

$$A = \{a^i b^j c^k \mid i = j \text{ or } j = k \text{ where } i, j, k \geq 0\}$$

Is your grammar ambiguous? Why or why not?
**Solution:** construct a CFG $G = (S, A, B, C, D\}, \{a, b, c\}, R, S)$ with the rules:

$$\begin{aligned} S &\to BC \mid AD \\ B &\to aBb \mid \epsilon \\ C &\to cC \mid \epsilon \\ A &\to aA \mid \epsilon \\ D &\to bDc \mid \epsilon \end{aligned}$$

Yes it is ambiguous. Choose the word $abc$. The two derivations are:

(a) $S \to BC \to aBbC \to abC \to abcC \to abc$,

(b) $S \to AD \to aAD \to aD \to abDc \to abc$.

2.18 a. Let $C$ be a context-free language and $R$ be a regular language. Prove that the language $C \cap R$ is context free.
**Solution:** answered in the text.

b. Let $A = \{w \mid w \in \{a, b, c\}^*$ and $w$ contains equal numbers of $a$?s, $b$?s, and $c$'s$\}$. Use part (a) to show that $A$ is not a CFL.
**Solution:** answered in the text.

2.24 Let $E = \{a^i b^j | i \neq j \text{ and } 2i \neq j\}$. Show that $E$ is a context-free language.
**Solution:** This is a partition of 3 languages: $L_1, L_2, L_3$ (i.e., it is their union). They are defined as follows:

- $L_1 = \{a^i b^j | i > j\}$,
- $L_2 = \{a^i b^j | i < j \text{ and } 2i > j\}$,
- $L_3 = \{a^i b^j | 2i < j\}$.

It suffices to show that $L_1, L_2, L_3$ are all context-free languages. They are due to the following grammars:

$L_1$:

- $S \to aSb|aA$,
- $A \to aA|\epsilon$

$L_2$:

- $S \to aAb$,
- $A \to aAb|aAbb|abb$.

$L_3$:

- $S \to aSbb|Ab$,
- $A \to Ab|\epsilon$

2.30 Use the pumping lemma to show that the following languages are not context free.

b. **Solution:** answered in the text.

c. **Solution:** answered in the text.

2.38 Refer to Problem 1.41 for the definition of the perfect shuffle operation. Show that the class of context-free languages is not closed under perfect shuffle.
**Solution:** answered in the text.

2.52 Show that every DCFG generates a prefix-free language.
**Solution:** answered in the text.

## 1.3 Chapter 3

3.1 This exercise concerns TM $M_2$, whose description and state diagram appear in Example 3.7. In each of the parts, give the sequence of configurations that $M_2$ enters when started on the indicated input string.

a. 0. **Solution:** $q_1 0 \to \sqcup q_2 \sqcup \to \sqcup\sqcup q_{accept}$.

b. 00. **Solution:** answered in the text.

c. 000. **Solution:** $q_1 000 \to \sqcup q_2 00 \to \sqcup x q_3 0 \to \sqcup x 0 q_4 \sqcup \to \sqcup x 0 \sqcup q_{reject}$.

d. 000000. **Solution:** $q_1 000000 \to \sqcup q_2 00000 \to \sqcup x q_3 0000 \sqcup \to \sqcup x 0 q_4 000 \to \sqcup x 0 x q_3 00 \to \sqcup x 0 x 0 q_4 0 \to \sqcup x 0 x 0 x q_3 \sqcup \to \sqcup x 0 x 0 q_5 x \sqcup \to \sqcup x 0 x q_5 0 x \sqcup \to \sqcup x 0 q_5 x 0 x \sqcup \to \sqcup x q_5 0 x 0 x \sqcup \to \sqcup q_5 x 0 x 0 x \sqcup \to q_5 \sqcup x 0 x 0 x \sqcup \to \sqcup q_2 x 0 x 0 x \sqcup \to \sqcup x q_2 0 x 0 x \sqcup \to \sqcup x x q_3 x 0 x \sqcup \to \sqcup x x x q_3 0 x \sqcup \to \sqcup x x x 0 q_4 x \sqcup \to \sqcup x x x 0 x q_4 \sqcup \to \sqcup x x x 0 x \sqcup q_{reject}$.

3.2 This exercise concerns TM $M_1$, whose description and state diagram appear in Example 3.9. In each of the parts, give the sequence of configurations that $M_1$ enters when started on the indicated input string.

a. 11. **Solution:** answered in the text.

b. 1#1. **Solution:** $q_1 1\#1 \to x q_3 \#1 \to x\# q_5 1 \to x q_6 \# x \to q_7 x\# x \to x q_1 \# x \to x\# q_8 x \to x\# q_8 \sqcup \to x\# x \sqcup q_{accept} \sqcup$.

c. 1##1. **Solution:** $q_1 1\#\#1 \to x q_3 \#\#1 \to x\# q_5 \#1 \to x\#\# q_{reject} 1$.

d. 10#11. **Solution:** $q_1 10\#11 \to x q_3 0\#11 \to x 0 q_3 \#11 \to x 0\# q_5 11 \to x 0 q_6 \# x 1 \to x q_7 0\# x 1 \to q_7 x 0\# x 1 \to x q_1 0\# x 1 \to x x q_2 \# x 1 \to x x\# q_4 x 1 \to x x\# x q_4 1 \to x x\# x 1 q_{reject} \sqcup$.

e. 10#10. **Solution:** $q_1 10\#10 \to x q_3 0\#10 \to x 0 q_3 \#10 \to x 0\# q_5 10 \to x 0 q_6 \# x 0 \to x q_7 0\# x 0 \to q_7 x 0\# x 0 \to x q_1 0\# x 0 \to x x q_2 \# x 0 \to x x\# q_4 x 0 \to x x\# x q_4 0 \to x x\# q_6 x x \to x x q_6 \# x x \to x x q_7 x\# x x \to x x q_7 x\# x x \to x x q_1 \# x x \to x x\# q_8 x x \to x x\# x x q_8 \sqcup \to x x\# x x \sqcup q_{accept} \sqcup$.

3.3 Modify the proof of Theorem 3.16 to obtain Corollary 3.19, showing that a language is decidable iff some nondeterministic Turing machine decides it. (You may assume the following theorem about trees. If every node in a tree has finitely many children and every branch of the tree has finitely many nodes, the tree itself has finitely many nodes.)
**Solution:** answered in the text.

3.5 Examine the formal definition of a Turing machine to answer the following questions, and explain your reasoning.
**Solution:** answered in the text.

3.6 In Theorem 3.21, we showed that a language is Turing-recognizable iff some enumerator enumerates it. Why didn?t we use the following simpler algorithm for the forward direction of the proof? As before, $s_1, s_2, \cdots$ is a list of all strings in $\Sigma^*$.
$E =$ "Ignore the input.

(a) Repeat the following for $i = 1, 2, 3, \cdots$.

(b) Run $M$ on $s_i$.

(c) If it accepts, print out si."

**Solution:** The second step might run forever, and therefore $E$ may not move onto the next string.

3.7 Explain why the following is not a description of a legitimate Turing machine.
$M_{bad} =$ "On input $\langle p \rangle$, a polynomial over variables $x_1, \cdots, x_k$:

(a) Try all possible settings of $x_1, \cdots, x_k$ to integer values.

(b) Evaluate $p$ on all of these settings.

(c) If any of these settings evaluates to 0, *accept*; otherwise, *reject*."

**Solution:** it's not a valid description because the machine will never halt, and the first step alone will take infinite time.

3.10 Say that a ***write-once Turing machine*** is a single-tape TM that can alter each tape square at most once (including the input portion of the tape). Show that this variant Turing machine model is equivalent to the ordinary Turing machine model. (Hint: As a first step, consider the case whereby the Turing machine may alter each tape square at most twice. Use lots of tape.)
**Solution:** answered in the text.

3.11 A ***Turing machine with doubly infinite tape*** is similar to an ordinary Turing machine, but its tape is infinite to the left as well as to the right. The tape is initially filled with blanks except for the portion that contains the input. Computation is defined as usual except that the head never encounters an end to the tape as it moves leftward. Show that this type of Turing machine recognizes the class of Turing-recognizable languages.
**Solution:** we simulate a doubly-infinite TM with an ordinary one with 2 tapes (which is equivalent to single-tape TMs). The first tape starts with the input string, and the second is blank. Cut the doubly-infinite tape into 2 parts, at the start of the input string, and these two parts are the same as the two tapes.

3.12 A ***Turing machine with left reset*** is similar to an ordinary Turing machine, but the transition function has the form

$$\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{\text{R, RESET}\}$$

If $\delta(q, a) = (r, b, \text{RESET})$, when the machine is in state $q$ reading an $a$, the machine?s head jumps to the left-hand end of the tape after it writes $b$ on the tape and enters state $r$. Note that these machines do not have the usual ability to move the head one symbol left. Show that Turing machines with left reset recognize the class of Turing-recognizable languages.
**Solution:** Let $M$ be a TM. We will construct a TM with left-reset $L$ as follows:
$L$ = "On input w:

(a) If $q$ is a halting (i.e., accept or reject) state, go to step d. Otherwise, execute Steps b or c depending on whether the current transition is right or left.

(b) If the current transition is right:

  i. For the current tape symbol $s$, and for $\delta(q, a) = (q', b, R)$, replace the $a$ with a $b$ and then RESET.

  ii. Scan right for a marked symbol - if there are none, RESET and mark the first symbol. Then, move the head to the right, change $L$'s state to $q'$, and go back to step a. If there is a marked symbol, move the tape head right.

  iii. Mark the symbol under the tape head, move right, and change $L$'s state to be $q'$. Then, go back to step a.

(c) If the current transition is left:

  i. If $\delta(q, a) = (q', b, L)$, replace the $a$ with $b$ and then RESET.

  ii. If the first symbol is marked, remove the mark, and then RESET. Also, change $L$'s state to be $q'$ and go back to step a.

  iii. Scan right for a marked symbol - if there are none, *reject* $w$. Otherwise, if one is found, RESET and mark the first symbol.

  iv. If the next symbol is marked, unmark it and RESET. Also, move right and change $L$'s state to be $q'$, and go back to step a.

  v. If the second symbol is not marked:

    A. RESET, move right to the first marked cell, and unmark it and move right again.

    B. Mark the current tape cell and move right again.

    C. If the current tape cell is unmarked, go back to A. Otherwise, unmark it and RESET.

D. Move right to the first marked cell. Move right and change $L$'s state to $q'$ and go back to step a.

(d) If $q'$ is an accept state, then *accept* $w$; otherwise, *reject* $w$."

3.15 Show that the collection of decidable languages is closed under the operation of

a. union. **Solution:** answered in the text.

3.16 Show that the collection of Turing-recognizable languages is closed under the operation of

a. union. **Solution:** answered in the text.

3.18 Show that a language is decidable iff some enumerator enumerates the language in the standard string order.
**Solution:** let the language be $L$. If $L$ is decidable, then we can just generate strings in lexicographic order, and test if each is in $L$, thus generating an enumerator that prints in standard string order. If we have such an enumerator $E$:

(a) If $L$ is finite, then just hardwire each of the strings in $L$ in $E$.

(b) If $L$ is infinite, then on an input $w$, run $E$ to print all strings in standard string order until a string lexicographically after $w$ appears; if $w$ has appeared, accept; otherwise, reject.

3.22 Let $A$ be the language containing only the single string $s$, where $s = 0$ if life never will be found on Mars, and $s = 1$ if life will be found on Mars someday. Is $A$ decidable? Why or why not? For the purposes of this problem, assume that the question of whether life will be found on Mars has an unambiguous YES or NO answer.
**Solution:** answered in the text.

## 1.4 Chapter 4

4.2 Consider the problem of determining whether a DFA and a regular expression are equivalent. Express this problem as a language and show that it is decidable.
**Solution:** We formulate the problem $EQ_{DFA,REX} = \{\langle A, R\rangle \mid A$ is a DFA, $R$ is a regular expression, and $L(A) = L(R)\}$. We will design a TM $T$ that decides $EQ_{DFA,REX}$:
$T =$ "On input $\langle A, R\rangle$ where $A$ is a DFA, $R$ is a regular expression:

1. Use Theorem 1.54 to convert $R$ into an equivalent DFA $B$. Therefore, $L(B) = L(R)$.
2. Run $EQ_{DFA}$ on input $\langle A, B\rangle$. Output what $EQ_{DFA}$ outputs."

Since $EQ_{DFA}$ is decidable, and the conversion from regular expressions to DFAs takes finite time, $EQ_{DFA,REX}$ is decidable.

4.3 Let $ALL_{DFA} = \{\langle A\rangle \mid A$ is a DFA and $L(A) = \Sigma^*\}$. Show that $ALL_{DFA}$ is decidable.
**Solution:** We will design a TM $T$ that decides $ALL_{DFA}$:
$T =$ "On input $\langle A\rangle$ where $A$ is a DFA:

1. Construct a DFA $B$ such that $L(B) = \overline{L(A)}$.
2. Run $E_{DFA}$ on input $\langle B\rangle$. Output what $E_{DFA}$ outputs."

Since $E_{DFA}$ is decidable, $ALL_{DFA}$ is decidable.

4.4 Let $A\epsilon_{CFG} = \{\langle G\rangle \mid G$ is a CFG that generates $\epsilon.\}$. Show that $A\epsilon_{CFG}$ is decidable.
**Solution:** We will design a TM $T$ that decides $A\epsilon_{CFG}$:
$T =$ "On input $\langle G\rangle$ where $G$ is a CFG:

1. Convert $G$ into an equivalent CFG $C = (V, \Sigma, R, S)$ in Chomsky Normal Form.
2. *Accept* $\langle G\rangle$ if $C$ includes the rule $S \to \epsilon$, *reject* $\langle G\rangle$ otherwise."

Since converting a CFG into CNF is decidable, $A\epsilon_{CFG}$ is decidable.

4.7 Let $\mathcal{B}$ be the set of all infinite sequences over $\{0,\ 1\}$. Shat that $\mathcal{B}$ is uncountable using a proof by diagonalization.
**Solution:** Suppose (by contradiction) that $\mathcal{B}$ is countable. Therefore, there exists a bijection $f$ between $\mathcal{B}$ and $\mathbb{N}$. For $\forall n \in \mathbb{N}$, let $f(n) = s_{n1}...s_{nk}$, where $s_{nk}$ is the $k$th bit in the $n$th sequence of $\mathcal{B}$ for $\forall k \in \mathbb{N}$. Define $t = t_1t_2...$ be an infinite sequence over $\{0,\ 1\}$ such that $t_k = |s_{kk} - 1|$ for $\forall k \in \mathbb{N}$. Therefore, $t \in \mathcal{B}$ by construction. We can see that $\nexists x \in \{0, 1\}$ such that $x = |1 - x|$. Therefore, for $\forall k \in \mathbb{N}, t_k \neq |s_{kk} - 1|$. This implies that in at least 1 bit, $t$ is different than all other sequences in $\mathcal{B}$ because of $t$'s construction, which involves all other sequences in $\mathcal{B}$. This implies for $\forall n \in \mathbb{N}, f(n) \neq t$, a contradiction. Therefore, $\mathcal{B}$ is uncountable.

4.10 Let $INFINITE_{DFA} = \{\langle M\rangle \mid M$ is a DFA and $L(M)$ is an infinite language$\}$. Show that $INFINITE_{DFA}$ is decidable.
**Solution:** answered in the text.

4.11 Let $INFINITE_{PDA} = \{\langle M\rangle \mid M$ is a PDA and $L(M)$ is an infinite language$\}$. Show that $INFINITE_{PDA}$ is decidable.
**Solution:** We will design a TM $T$ that decides $INFINITE_{PDA}$:
$T =$ "On input $\langle M\rangle$ where $M$ is a PDA:

1. Construct an equivalent CFG $G$ from $M$.
2. Convert $G$ into an equivalent CFG $C = (V, \Sigma, R, S)$ in Chomsky Normal Form.
3. *Accept* $\langle M\rangle$ if there exists a derivation $A \overset{+}{\Rightarrow} uAv$ for some $u, v \in \Sigma^*$. Otherwise, *reject* $\langle M\rangle$.

Since all of the algorithms in this machine are decidable, $INFINITE_{PDA}$ is decidable.

4.12 Let $A = \{\langle M\rangle \mid M$ is a DFA that doesn't accept any string containing an odd number of 1s$\}$. Show that $A$ is decidable.
**Solution:** answered in the text.

4.13 Let $A = \{\langle R, S\rangle \mid R$ and $S$ are regular expressions and $L(R) \subseteq L(S)\}$. Show that $A$ is decidable.
**Solution:** We will design a TM $T$ that decides $A$:
$T =$ "On input $\langle R, S\rangle$ where $R$ and $S$ are regular expressions:

1. Construct a DFA $B$ such that $L(B) = \overline{L(S)} \cap L(R)$.
2. Run $E_{DFA}$ on input $\langle B\rangle$. Output what $E_{DFA}$ outputs."

Since $E_{DFA}$ is decidable, $A$ is decidable. This construction is correct because $L(R) \subseteq L(S) \Leftrightarrow \overline{L(S)} \cap L(R) = \varnothing$.

4.14 Let $\Sigma = \{0, 1\}$. Show that the problem of determining whether a CFG generates some string in $1^*$ is decidable. In other words, show that

$$\{\langle G\rangle \mid G \text{ is a CFG over } \{0,\ 1\} \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

is a decidable language.
**Solution:** answered in the text.

4.15 Show that the problem of determining whether a CFG generates all strings in 1* is decidable. In other words, show that $\{\langle G\rangle \mid G$ is a CFG over $\{0, 1\}$ and $1^* \subset L(G)\}$ is a decidable language.
**Solution:** Let $f$ be a computable function. Construct a decider $D$:
$D =$ "On input $\langle G\rangle$ where $G$ is a CFG:

1. Convert $G$ into an equivalent CFG $C$ in Chomsky Normal Form.
2. Let $p$ be the pumping length of $C$.
3. Repeat $0 \leq i \leq p + p!$:
   a. Check whether $1^i \in L(C)$.
   b. If not, *reject* $\langle G\rangle$.
4. *Accept* $\langle G\rangle$.

We can check $1^i \in L(C)$ using the Cocke-Younger-Kasami (CYK) algorithm for CFGs, which has a running time of $\Theta(n^3|G|)$. Therefore, the loop has a running time of $\Theta(pn^3|G|)$. Since Steps 1, 2, 3, and 4 take finite time, this language is decidable.

4.23 Say that an NFA is ambiguous if it accepts some string along two different computation branches. Let $AMBIG_{NFA} = \{\langle N\rangle \mid N$ is an ambiguous NFA$\}$. Show that $AMBIG_{NFA}$ is decidable. (Suggestion: One elegant way to solve this problem is to construct a suitable DFA and then run $E_{DFA}$ on it.)
**Solution:** answered in the text.

4.24 A ***useless state*** in a pushdown automaton is never entered on any input string. Consider the problem of determining whether a pushdown automaton has any useless states. Formulate this problem as a language and show that it is decidable.
**Solution:** Let $U_{PDA} = \{\langle P\rangle \mid P$ is a PDA that has useless states$\}$. We will design a TM $T$ that decides $U_{PDA}$. First, we will define the problem $E_{PDA} = \{\langle P\rangle \mid P$ is a PDA and $L(P) = \varnothing\}$. We now show that $E_{PDA}$ is decidable with a decider $D$:
$D =$ "On input $\langle P\rangle$ where $P$ is a PDA:

1. Convert $P$ into an equivalent CFG $G$.
2. Run $E_{CFG}$ on input $\langle G\rangle$. Output what $E_{CFG}$ outputs."

This language is decidable because all steps in its construction take finite time, and $E_{CFG}$ is a decider.
$T =$ "On input $\langle P\rangle$ where $P = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ is a PDA:

1. For $\forall q \in Q$:
   a. Modify $P$ such that $F = \{q\}$.
   b. Run $E_{PDA}$ on input $\langle P \rangle$. If $E_{PDA}$ accepts, *accept* $\langle P \rangle$.
2. *Reject* $\langle P \rangle$."

Since $E_{PDA}$ is decidable, $U_{PDA}$ is decidable.

4.25 Let $BAL_{DFA} = \{\langle M \rangle \mid M$ is a DFA that accepts some string containing an equal number of 0s and 1s$\}$. Show that $BAL_{DFA}$ is decidable. (Hint: Theorems about CFLs are helpful here.)
**Solution:** answered in the text.

## 1.5 Chapter 5

5.5 Show that $A_{TM}$ is not mapping reducible to $E_{TM}$. In other words, show that no computable function reduces $A_{TM}$ to $E_{TM}$. (Hint: Use a proof by contradiction, and facts you already know about $A_{TM}$ and $E_{TM}$.)
**Solution:** answered in the text.

5.6 Show that $\leq_m$ is a transitive relation.
**Solution:** answered in the text.

5.7 Show that if $A$ is Turing-recognizable and $A \leq_m \overline{A}$, then $A$ is decidable.
**Solution:** answered in the text.

5.8 In the proof of Theorem 5.15, we modified the Turing machine $M$ so that it never tries to move its head off the left-hand end of the tape. Suppose that we did not make this modification to $M$. Modify the PCP construction to handle this case.
**Solution:** answered in the text.

5.10 Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape when it is run on input $w$. Formulate this problem as a language and show that it is undecidable.
**Solution:** answered in the text.

5.11 Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.
**Solution:** answered in the text.

5.13 A ***useless state*** in a Turing machine is one that is never entered on any input string. Consider the problem of determining whether a Turing machine has any useless states. Formulate this problem as a language and show that it is undecidable.
**Solution:** Let $USELESS_{TM} = \{\langle M, q\rangle \mid q \text{ is a useless state in } M\}$. Suppose that $USELESS_{TM}$ was decidable, and let $U$ be its decider. We will construct a decider $E$ for $E_{TM}$:
$E =$ "On input $\langle M\rangle$ where $M$ is a TM:

1. Run $U$ on $\langle M, q_{accept}\rangle$, where $q_{accept}$ is $M$'s accept state.
2. Output what $U$ outputs.

Since $E_{TM}$ is undecidable, $USELESS_{TM}$ is also undecidable.

5.17 Show that the Post Correspondence Problem is decidable over the unary alphabet $\Sigma = \{1\}$.
**Solution:** Since $\Sigma = \{1\}$, there is only a difference in the number of 1s on the top of each domino compared to the bottom. We will construct a decider $D$ that decides PCP for a unary alphabet:
$D =$ "On input a collection of dominos:

1. If any domino has the same number of 1s on the top and bottom, *accept*.
2. If all dominos have more 1s on the top than the bottom, or more 1s on the bottom than the top, *reject*.
3. Let $d_1$ be one domino with $c_1$ more 1s on the top than the bottom, and $d_2$ be another domino with $c_2$ more 1s on the bottom than the top.
4. Choose $c_2$ of the $d_1$ domino, and $c_1$ of the $d_2$ domino.

Step 4 guarantees a match. Let $d_1$ have $t_1$ 1s on the top, and $t_1 - c_1$ 1s on the bottom. Let $d_2$ have $t_2$ 1s on the top and $t_2 + c_2$. Choosing $c_2$ of the $d_1$ domino and $c_1$ of the $d_2$ domino yields $c_2 \times t_1 + c_1 \times t_2$ 1s on the top, and $c_2 \times (t_1 - c_1) + c_1 \times (t_2 + c_2) = c_2 \times t_1 + c_1 \times t_2$ 1s on the bottom, which is a match.

5.24 Let $J = \{w|$ either $w = 0x$ for some $x \in A_{TM}$, or $w = 1y$ for some $y \in \overline{A_T M}\}$. Show that neither $J$ nor $\overline{J}$ is Turing-recognizable.
**Solution:** let $L_1 = \{\langle M, x\rangle | M$ is a TM and $M$ does not accept $x\}$. We can see $\overline{A_{TM}} \leq_m L_1$. We show $L_1 \leq_m J$ with the following reduction: on input $w$, output $1w$. We have that $w \in L_1$ if and only if $1w \in J$, showing $J$ is not Turing-recognizable. We now show $\overline{A_{TM}} \leq_m \overline{J}$, done by the following reduction: on input $w$, output $0w$. We have $w \in \overline{A_{TM}}$ if and only if $0w \in \overline{J}$, showing that $\overline{J}$ is not Turing-recognizable.

5.25 Give an example of an undecidable language $B$, where $B \leq_m \overline{B}$.
**Solution:** Any Turing-recognizable but not co-Turing-recognizable language works (or vice versa), such as $A_{TM}$.

5.26 Define a ***two-headed finite automaton*** (2DFA) to be a deterministic finite automaton that has two read-only, bidirectional heads that start at the left-hand end of the input tape and can be independently controlled to move in either direction. The tape of a 2DFA is finite and is just large enough to contain the input plus two additional blank tape cells, one on the left-hand end and one on the right-hand end, that serve as delimiters. A 2DFA accepts its input by entering a special accept state. For example, a 2DFA can recognize the language $\{a^n b^n c^n | n \geq 0\}$.

a. Let $A_{2DFA} = \{\langle M, x\rangle | M$ is a 2DFA and $M$ accepts $x\}$. Show that $A_{2DFA}$ is decidable.
**Solution:** since the input tape is only finitely long, there are only finitely many different configurations that the 2DFA can be in: if it has $|Q|$ states with input $w$ of length $|w|$ (counting the delimiters), the number of configurations is $|Q| \times |w|^2$. We can decide $A_{2DFA}$ with a TM $T$ as follows:
$T =$ "On input $\langle M, x\rangle$ where $M$ is a 2DFA and $x$ is a string, and $M$ has states $Q$:
   i. Simulate $M$ on $x$ for $|Q| \times |x|^2$ steps.
   ii. If $M$ halts within $|Q| \times |x|^2$ steps, then *accept* $x$; otherwise, *reject* $x$."

b. Let $E_{2DFA} = \{\langle M\rangle | M$ is a 2DFA and $L(M) = \emptyset\}$. Show that $E_{2DFA}$ is not decidable
**Solution:** we show $E_{2DFA} \leq_m A_{TM}$. Given $\langle M\rangle$ and $w$, we can create a 2DFA that accepts all accepting computation histories for $M$ on $w$, and rejects all other strings. Then we just test if the language is empty. The proof goes the same way as did for $E_{LBA}$.

5.28 **Rice's theorem**. Let $P$ be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property $P$ is undecidable.

In more formal terms, let $P$ be a language consisting of Turing machine descriptions where $P$ fulfills two conditions. First, $P$ is nontrivial–it contains some, but not all, TM descriptions. Second, $P$ is a property of the TM's language–whenever $L(M_1) = L(M_2)$, we have $\langle M_1\rangle \in P$ iff $\langle M_2\rangle \in P$. Here, $M_1$ and $M_2$ are any TMs. Prove that $P$ is an undecidable language.
**Solution:** answered in the text.

5.29 Show that both conditions in Problem 5.28 are necessary for proving that P is undecidable.
**Solution:** If the property $P$ is trivial, then one of the following is true:

1. All TM descriptions are in the language (if p includes all TMs)
2. The language is empty.

For 1, we just build a TM that accepts if $M$ is a valid TM description. In the latter case, we build a TM that rejects all strings. If $P$ is a property of the machine rather than the language, it is possible for the language to be decidable.

5.30 Use Rice's theorem, which appears in Problem 5.28, to prove the undecidability of each of the following languages.

a. $INFINITE_{TM} = \{\langle M\rangle \mid M$ is a TM and $L(M)$ is an infinite language$\}$.
**Solution:** answered in the text.

b. $\{\langle M \rangle \mid M \text{ is a TM and } 1011 \in L(M)\}$.
**Solution:** Let $P$ be this language. We can clearly see $P$ is a language of descriptions of TMs. It is non-trivial because some TMs contain the string 1011 in their language, and others do not. Also, it only depends on the language. If two TMs recognize the same language, either both have their descriptions in $P$, or neither do. Therefore, we can apply Rice's theorem and conclude that $P$ is undecidable.

c. $ALL_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$.
**Solution:** $ALL_{TM}$ is non-trivial because some TMs accept $\Sigma^*$ and others do not. Also, it only depends on the language. If two TMs recognize $\Sigma^*$, then the descriptions of both are in $ALL_{TM}$ or neither are. Therefore, we can apply Rice's theorem and conclude that $ALL_{TM}$ is undecidable.

## 1.6 Chapter 6

6.6 Describe two different Turing machines, $M$ and $N$, where $M$ outputs $\langle N \rangle$ and $N$ outputs $\langle M \rangle$, when started on any input.
**Solution:** Construct $M$ as follows:
$M =$ "On input $w$:

1. Obtain $\langle M \rangle$ by the recursion theorem.
2. Compute $q(\langle M \rangle)$, and write the result to the tape.
3. Halt."

Now let $N = q(\langle M \rangle)$. The computation in Step 2 writes $\langle q(\langle M \rangle) \rangle = \langle N \rangle$. $N$ writes $\langle M \rangle$ to the tape, so the construction of $M$ and $N$ is correct.

6.9 Use the recursion theorem to give an alternative proof of Rice?s theorem in Problem 5.28.
**Solution:** answered in the text.

6.10 Give a model of the sentence $\phi_{eq} = \forall x[R_1(x,x)] \wedge \forall x,y[R_1(x,y) \leftrightarrow R_1(y,x)] \wedge \forall x,y,z[(R_1(x,y) \wedge R_1(y,z)) \rightarrow R_1(x,z)]$.
**Solution:** answered in the text.

6.12 Let $(\mathbb{N}, <)$ be the model with universe $\mathbb{N}$ and the "less than" relation. Show that $\mathrm{Th}(\mathbb{N}, <)$ is decidable.
**Solution:** answered in the text.

6.13 For each $m > 1$ let $\mathbb{Z}_m = \{0, 1, 2, \cdots, m-1\}$, and let $F_m = (\mathbb{Z}_m, +, \times)$ be the model whose universe is $\mathbb{Z}_m$ and that has relations corresponding to the $+$ and $\times$ relations computed modulo $m$. Show that for each $m$, the theory $\mathrm{Th}(F_m)$ is decidable.
**Solution:** we are given a formula $\phi = Q_1 x_1 \cdots Q_k x_k \psi(x_1, \cdots, x_n)$ (assuming the $Q_i$ are quantifiers, and $\psi$ has no quantifiers). We iteratively define $I_\ell$ for $0 \leq \ell \leq n$:

- $I_n(x_1, \cdots, x_n) = \psi(x_1, \cdots, x_n)$,
- $I_{\ell-1}(x_1, \cdots, x_{\ell-1}) = \bigvee_{i=0}^{m-1} I_i(x_1, \cdots, x_{\ell-1}, i)$ if $Q_\ell$ is $\exists$,
- $I_{\ell-1}(x_1, \cdots, x_{\ell-1}) = \bigwedge_{i=0}^{m-1} I_i(x_1, \cdots, x_{\ell-1}, i)$ if $Q_\ell$ is $\forall$.

We then output the value of $I_0$. By an induction argument, we correctly calculate the truth value of $\phi$. Therefore, $\mathrm{Th}(F_m)$ is decidable.

6.23 Show that the function $K(x)$ is not a computable function.
**Solution:** Suppose that $K(x)$ were computable. We now construct a TM $Z$:
$Z =$ "On any input:

1. Obtain $\langle Z \rangle$ by the recursion theorem.
2. Compute $d = K(\langle Z \rangle)$.
3. Enumerate strings $x \in \Sigma^*$ in any order until $K(x) > d$.
4. Output $x$."

This a contradiction since we are outputting a value of $K(x)$ that is greater than the own Turing Machine's value. Therefore, $K(x)$ is not computable.

6.24 Show that the set of incompressible strings is undecidable.
**Solution:** Since the set of incompressible strings is an infinite set, this reduces to 6.25.

6.25 Show that the set of incompressible strings contains no infinite subset that is Turing-recognizable.
**Solution:** Suppose that such a subset were Turing-recognizable, and let $E$ be the enumerator, since Turing-enumerable languages are exactly the Turing-recognizable ones. We now construct a TM $Y$:
$Y =$ "On any input:

1. Obtain $\langle Y \rangle$ by the recursion theorem.

2. Run $E$ until it outputs a string $x$ with $|x| > |\langle Y \rangle|$.
3. Output $x$."

This a contradiction - therefore, no infinite subset of the set of incompressible strings is Turing-recognizable.

6.27 Let $S = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \{\langle M \rangle\}\}$. Show that neither $S$ nor $\overline{S}$ is Turing-recognizable.
**Solution:** We will show that $A_{TM} \leq_m S$ and $A_{TM} \leq_m \overline{S}$, implying that neither $S$ nor $\overline{S}$ is Turing-recognizable. For $A_{TM} \leq_m S$, given $\langle M, w \rangle \in A_{TM}$, construct a TM $T$ that, on input $x$, rejects if $x \neq \langle T \rangle$ and simulates $M$ on $w$ otherwise. Therefore, $L(T) = \{\langle T \rangle\}$ if $M$ accepts $w$, and $\emptyset$ otherwise.

For $A_{TM} \leq_m \overline{S}$, change $T$'s construction to be "accepts" instead of "rejects".

## 1.7 Chapter 7

7.1 Answer each part TRUE or FALSE.

c. **Solution:** answered in the text.

d. **Solution:** answered in the text.

7.2 Answer each part TRUE or FALSE.

c. **Solution:** answered in the text.

d. **Solution:** answered in the text.

7.15 Show that P is closed under the star operation. (Hint: Use dynamic programming. On input $y = y_1 \cdots y_n$ for $y_i \in \Sigma$, build a table indicating for each $i \leq j$ whether the substring $y_i \cdots y_j \in A^*$ for any $A \in$ P.)
**Solution:** Let $M$ be a TM that decides $A$ in polynomial time. We construct a TM $T$ to decide $A^*$ in polynomial time:

$T =$ "On input $w \in \Sigma^*$:

1. Let $|w| = n \in \mathbb{N}$.
2. If $n = 0$, *accept* $w$.
3. Construct an $n \times n$ table $B$, initializing all elements to 0.
4. For $1 \leq i \leq n$ do:
   a. If $w_i \in A$, set $B(i, i) = 1$.
5. For $2 \leq j \leq n$ do:
   a. For $1 \leq k \leq n - j + 1$ do:
      i. Set $m \leftarrow j + k - 1$.
      ii. If $w_k \cdots w_m \in A$, set $B(k, k) = 1$.
      iii. For $k \leq p \leq m - 1$ do:
         1. If $B(k, p) = B(p, m) = 1$, set $B(k, m) = 1$.
6. Output if $B(1, n) = 1$."

7.16 Show that NP is closed under the star operation.
**Solution:** answered in the text.

7.20 We generally believe that $PATH$ is not NP-complete. Explain the reason behind this belief. Show that proving $PATH$ is not NP-complete would prove P $\neq$ NP.
**Solution:** We can prove the converse: if P = NP, then $PATH$ is NP-complete. We will do two things (in assuming P = NP): show that $PATH$ is NP-hard, and because $PATH \in$ NP, $PATH$ is NP-complete. We will do this by showing $SAT \leq_p PATH$.

Since P = NP, there is a decider $S$ for $SAT$ that runs in time $O(n^k)$ for some $k \in \mathbb{N}$. We construct a TM $T$ that computes the reduction:

$T =$ "On input $\langle \phi \rangle$ where $\phi$ is a $SAT$ formula:

1. Run $S$ on input $\langle \phi \rangle$. If $S$ accepts $\langle \phi \rangle$, the construct a graph $G = (V, E)$ such that $s, t \in V, (s, t) \in E$.
2. If $S$ rejects $\langle \phi \rangle$, construct a graph $G = (V, E)$ such that $V = \{s, t\}, E = \emptyset$.
3. Output $\langle G, s, t \rangle$."

Since this reduction takes polynomial time, then P = NP implies $PATH$ is NP-complete.

7.22 Let $DOUBLE\text{-}SAT = \{\langle\phi\rangle \mid \phi$ has at least two satisfying assignments$\}$. Show that $DOUBLE\text{-}SAT$ is NP-complete.
**Solution:** We will reduce $3SAT$ to this problem. Given $\phi$ as a $3SAT$ formula, all we need to do is to introduce a new variable $x$, and create a new formula $\phi'$ as follows:

$$\phi' = \phi \wedge (x \vee \overline{x})$$

We show that $\phi \in 3SAT$ if and only if $\phi' \in DOUBLE\text{-}SAT$. If $\phi \in 3SAT$, then there is one satisfying assignment - for $\phi'$ there are two since we can set $x = 0, 1$ and still have $\phi'$ be true. If $\phi$ is unsatisfiable, then clearly $\phi'$ is also unsatisfiable. Therefore, we are done.

7.23 Let $HALF\text{-}CLIQUE = \{\langle G\rangle \mid G$ is an undirected graph having a complete subgraph with at least $\frac{m}{2}$ nodes, where $m$ is the number of nodes in $G\}$. Show that $HALF\text{-}CLIQUE$ is NP-complete.
**Solution:** answered in the text.

7.33 In the following solitaire game, you are given an $m \times m$ board. On each of its $m^2$ positions lies either a blue stone, a red stone, or nothing at all. You play by removing stones from the board until each column contains only stones of a single color and each row contains at least one stone. You win if you achieve this objective. Winning may or may not be possible, depending upon the initial configuration. Let $SOLITAIRE = \{\langle G\rangle \mid G$ is a winnable game configuration$\}$. Prove that $SOLITAIRE$ is NP-complete.
**Solution:** answered in the text.

7.38 Show that if P = NP, a polynomial time algorithm exists that produces a satisfying assignment when given a satisfiable Boolean formula. (Note: The algorithm you are asked to provide computes a function; but NP contains languages, not functions. The P = NP assumption implies that SAT is in P, so testing satisfiability is solvable in polynomial time. But the assumption doesn?t say how this test is done, and the test may not reveal satisfying assignments. You must show that you can find them anyway. Hint: Use the satisfiability tester repeatedly to find the assignment bit-by-bit.)
**Solution:** The assumption of P = NP implies SAT $\in$ P with polynomial-time TM $M$. We will construct a polynomial-time TM $B$ that outputs, given $\phi$ an assignment if one exists:
$B =$ "On input boolean formula $\phi$ over variables $x_1, \cdots, x_k$:

(a) Run $M$ on $\phi$. If $M$ rejects, *reject* $\phi$. Otherwise, continue.

(b) For $1 \leq i \leq k$:

   i. Let $\phi'$ be $\phi$ but with the variable $x_i$ hard-coded to 0.

   ii. Run $M$ on $\phi'$. If $M$ accepts, overwrite all instances of $x_i$ in $\phi$ to be 0; otherwise, overwrite all instances of $x_i$ in $\phi$ to be 1.

$B$ runs in polynomial time because setting the $x_i$ takes $O(|\phi|)$ time for each $x_i$. Since there are $O(|\phi|)$ variables, and calling $M$ takes polynomial time by assumption, $B \in$ P. The logic is correct because the first step checks whether $\phi$ is satisfiable or not - when we use the for loop, we know that an assignment exists. If setting $x_i$ to be 0 makes $M$ reject $\phi$, then the setting of $x_i$ must be 1.

7.40 Show that if P = NP, a polynomial time algorithm exists that takes an undirected graph as input and finds a largest clique contained in that graph. (See the note in Problem 7.38.)
**Solution:** answered in the text.

7.49 Let $f : \mathcal{N} \to \mathcal{N}$ be any function where $f(n) = o(n \log n)$. Show that $TIME(f(n))$ contains only the regular languages.
**Solution:** We define a ***crossing sequence*** (abbreviated "C.S.") for a TM $M$ as the sequence of states that $M$ enters at a given tape cell during computation on an input. We will prove two things, which is sufficient to prove that $TIME(f(n))$ contains only the regular languages:

1. A deterministic, single-tape TM with C.S.'s of bounded length decides a regular language.
2. A deterministic, single-tape TM with C.S.'s of unbounded length cannot run in $o(n \log n)$ time.

For the first part, let $M$ be a deterministic, single-tape TM, and consider $M$ running on some input. If all of $M$'s C.S.'s on these inputs have length $\leq k$, we can construct an NFA $N$ that simulates $M$, because $N$ only needs enough states to record the current C.S. plus a start state. $N$ then guesses the sequence of C.S.'s that would occur if $M$ processed the input string further, and also checks that adjacent C.S.'s are consistent (somewhat like a "computation history"). Without loss of generality, assume that we modify $M$ so that it accepts at the right-most end of the input. Also, $N$'s accept states correspond to C.S.'s that contain $M$'s accept state and that match with $M$'s computation on the blank portion of the tape (initially) after the input string. Therefore, a deterministic, single-tape TM with C.S.'s of bounded length decides a regular language.

For the second part, we will prove by contradiction. Suppose (to the contrary) that we have a deterministic, single-tape TM $M$ that runs in $o(n \log n)$ time and has C.S.'s of unbounded length. We can easily see that $M$ runs in time $bn\log(n)$, for all $n \geq n_0$ for some constant $n_0$ (also note that all inputs are of length $n$). We can also see that for all possible inputs, at least half of the C.S.'s must have length $\leq 2b\log(n)$, as not to exceed M's running time. Define these C.S.'s as the ***short crossing sequences*** (abbreviated "S.C.S."). If $M$ has $|Q|$ states, the number of distinct C.S.'s of length $p$ is $|Q|^p$. If we choose $b$ to be small enough, it is possible to force repetition among the S.C.S.'s because $|Q|^{2b\log(n)}$ is much less than $\frac{n}{2}$. For $\forall k \in \mathbb{N}$, consider inputs that have a C.S. of length $\geq k$ and that are of length $n_0$ or more. Let $w$ be the shortest such string, and let $i$ be the position in $w$ such that the longest C.S. occurs. We can see that 3 positions $(p_1, p_2, p_3)$ must exist that have exactly the same C.S.'s. Remove either the substring $p_1$ through $p_2 - 1$ or $p_2$ through $p_3 - 1$ from $w$ such that position $i$ is not removed. We can also see that $M$ runs on the modified string exactly the same way as it originally did on $w$, but now except for the removed portion, because positions of $w$ with exactly the same C.S.'s were overlaid. Therefore, there exists a C.S. of length $\geq k$ in this modified input. However, we originally selected $w$ to be the shortest string which has a C.S. of length $\geq k$, a contradiction. Therefore, a deterministic, single-tape TM with C.S's of unbounded length cannot run in $o(n \log n)$ time.

This is quite strange, as all regular languages are in $TIME(n)$, and there are not any in $TIME(f(n)) \cap \overline{TIME(n)}$ where $f(n) \in o(n \log n)$. However, according to (`http://arxiv.org/pdf/1307.3648.pdf`), any TM running in time $f(n) \in o(n \log n)$ actually runs in linear time.

## 1.8 Chapter 8

8.6 Show that any PSPACE-hard language is also NP-hard.
**Solution:** by definition of PSPACE-hardness for a language $L$, for all $A \in$ PSPACE, $A \leq_P L$. Since SAT $\in$ NP, SAT $\in$ PSPACE because NP $\subseteq$ PSPACE. Therefore, SAT $\leq_P L$. Since SAT is NP-complete, it is NP-hard. Therefore since SAT $\leq_P L$, $L$ is NP-hard (by transitivity of $\leq_P$).

## 1.9 Chapter 9

## 1.10 Chapter 10

10.19 Show that if $\mathsf{NP} \subseteq \mathsf{BPP}$, then $\mathsf{NP} = \mathsf{RP}$.

**Solution:** Since $\mathsf{RP} \subseteq \mathsf{NP}$, we only need to show $\mathsf{NP} \subseteq \mathsf{RP}$. So assume $\mathsf{NP} \subseteq \mathsf{BPP}$. Therefore, $\mathsf{SAT} \in \mathsf{BPP}$, and since it is $\mathsf{NP}$-complete, we only need to show it is in $\mathsf{RP}$.

Let $M$ be a probabilistic poly-time TM that accepts $\mathsf{SAT}$ with error at most $\frac{1}{3}$.

## Problem

Examine the following formal descriptions of sets so that you understand which members they contain. Write a short informal English description of each set.

**a.** {1, 3, 5, 7, . . . }

**b.** { . . . ,–4,–2, 0, 2, 4, . . . }

**c.** {nl n = 2m for some m in N}

**d.** {nl n = 2m for some m in N, and n = 3k for some k in N}

**e.** {wl w is a string of 0s and 1s and w equals the reverse of w}

**f.** {nl n is an integer and n = n + 1}

## Step-by-step solution

**Step 1** of 5

a)

A set of all odd natural numbers.

b)

A set of all even integers.

Comment

**Step 2** of 5

c)

A set of all even natural numbers.

(or) A set of all natural numbers divisible by 2.

Comment

**Step 3** of 5

d)

A set of all natural numbers, divisible by both 2 and 3.

(or) A set of all natural multiples of 6.

Comment

**Step 4** of 5

e)

The set of all strings comprising of 0's and 1's and every string is a palindrome.

Comment

**Step 5** of 5

f)

The set of all integers that are equal to one added to that number.

## Problem

Write formal descriptions of the following sets.

**a.** The set containing the numbers 1, 10, and 100

**b.** The set containing all integers that are greater than 5

**c.** The set containing all natural numbers that are less than 5

**d.** The set containing the string aba

**e.** The set containing the empty string

**f.** The set containing nothing at all

## Step-by-step solution

**Step 1** of 6

**a.**

Consider the given information:

The set containing the number of series of 10. It means n contain 10 and 10 to the power of m, contain the series of 0, 1, and 2, is as shown below:

$$\{n : n = 10^m \text{ for some } m \in \{0,1,2\}\}$$

Comment

**Step 2** of 6

**b.**

Consider the given information:

The set contains the all integer number which is greater than 5.

Thus, the n which contains only those integer numbers which are greater than 5 is as shown below:

$$\{n : n \text{ is an integer and } n > 5\}$$

Comment

**Step 3** of 6

**c.**

Consider the given information:

The set contains the all natural number which is less than 5.

Thus, the n which contains only those natural numbers which are less than 5 is as shown below:

$$\{n : n \text{ is a natural number and } n < 5\}$$

Comment

**Step 4** of 6

**d.**

Consider the given information:

The set contains the string which is aba.

Thus, the n which contains only the string 'aba', is as shown below:

$$\{\text{n : n contains } \{\text{aba}\}\}.$$

Comment

**Step 5** of 6

**e.**

Consider the given information:

The set contains the empty string. Empty is denoted by $\varepsilon$.

Thus, the set which contains only the empty string, is as shown below:

$\{\varepsilon\}$

Comment

**Step 6** of 6

**f.**

Consider the given information:

The set contains nothing, it means set contain only null. Null is denoted by $\phi$.

Thus, the set which does not contains anything is as shown below:

$\varnothing$

Comment

## Problem

Let A be the set {x, y, z} and B be the set {x, y}.

**a.** Is A a subset of B?

**b.** Is B a subset of A?

**c.** What is A ∪ B?

**d.** What is A ∩ B?

**e.** What is A × B?

**f.** What is the power set of B?

## Step-by-step solution

**Step 1** of 6

(a) No, $A$ is not a subset of $B$. $A$ contains an extra element z which is not there in $B$.

Comment

**Step 2** of 6

(b) Yes, $B$ is a subset of $A$, because every member of $B$ also is a member of $A$. In fact $B$ is a proper subset of $A$ in this case.

Comment

**Step 3** of 6

(c)

$$\begin{aligned} A \cup B &= \{x, y, z\} \cup \{x, y\} \\ &= \{x, y, z\} \\ &= A \end{aligned}$$

Comment

**Step 4** of 6

(d)

$$\begin{aligned} A \cap B &= \{x, y, z\} \cap \{x, y\} \\ &= \{x, y\} \\ &= B \end{aligned}$$

Comment

**Step 5** of 6

(e)

$$\begin{aligned} A \times B &= \{(a, b) : a \in A \text{ and } b \in B\} \\ &= \{(x, x), (x, y), (y, x), (y, y), (z, x), (z, y)\} \end{aligned}$$

Comment

**Step 6** of 6

(f) Power set of B is {∅, {x}, {y}, {x, y}}

## Problem

If A has a elements and B has b elements, how many elements are in A × B? Explain your answer.

## Step-by-step solution

**Step 1** of 1

$A \times B$ will have $a \times b$ number of elements in it.

One of the method of constructing the Cartesian product is to select an element of $A(x_a)$ and pair it with each and every element of *B:* $B(y_1, y_2 ... y_b)$. This produces the pairings $\{(x_1, y_1), (x_1, y_2), ..., (x_1, y_b)\}$. On repeating this procedure for each remaining element of $A(x_2)$ through $x_a$.

First pairing will produce $\sum_{i=1}^{n}(x_1, y_1) = a$ pairs. As the iteration continues over the elements of *A*, *a* number of sets will get generated, each set having *b* pairs.

Thus, the number of elements in $A \times B$ is $a \times b.$

Comment

## Problem

If C is a set with c elements, how many elements are in the power set of C? Explain your answer.

## Step-by-step solution

**Step 1** of 1

Consider the data, Number of elements in set $C = c$

**Formula:**

When the number of elements in the set S is *n*, then its power set consists of $2^n$ elements.

The Power set is the set of all subsets of the set *S*.

The set *C* contains *c* elements such as $\{c_1, c_2, c_3, ....c_c\}$. Substituting, '*c*' instead of '*n*'. The number of elements in the power set of C is ' $2^c$' elements.

**Example:**

Assume $C = \{1,2,3,4\}$ .So, the number of elements in set *C* is 4.

The total subsets of the set *C* are:

$$\{\ \},\{1\},\{2\},\{3\},\{4\},\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},$$
$$\{3,4\},\{1,2,3\},\{1,2,4\},\{1,3,4\},\{2,3,4\},\{1,2,3,4\}.$$

The set of all subsets is called a Power set. Thus, the power set is as follows:

$$P(C) = \left\{ \begin{array}{l} \{\ \},\{1\},\{2\},\{3\},\{4\},\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\}, \\ \{3,4\},\{1,2,3\},\{1,2,4\},\{1,3,4\},\{2,3,4\},\{1,2,3,4\} \end{array} \right\}$$

The number of elements in the power set is 16 $\left(2^4\right)$.

Thus, when the number of elements in a set *C* is *c*, then the number of elements in its power set consists of $2^c$ elements.

Comment

## Problem

Let X be the set {1, 2, 3, 4, 5} and Y be the set {6, 7, 8, 9, 10}. The unary function f : X → Y and the binary function g : X ×Y →Y are described in the following tables.

| $n$ | $f(n)$ |
|---|---|
| 1 | 6 |
| 2 | 7 |
| 3 | 6 |
| 4 | 7 |
| 5 | 6 |

| $g$ | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| 1 | 10 | 10 | 10 | 10 | 10 |
| 2 | 7 | 8 | 9 | 10 | 6 |
| 3 | 7 | 7 | 8 | 8 | 9 |
| 4 | 9 | 8 | 7 | 6 | 10 |
| 5 | 6 | 6 | 6 | 6 | 6 |

**a.** What is the value of f(2)?

**b.** What are the range and domain of f?

**c.** What is the value of g(2, 10)?

**d.** What are the range and domain of g?

**e.** What is the value of g(4, f(4))?

## Step-by-step solution

**Step 1** of 5

**a)**

The value of the function *f* at n is 2 is 7.

Therefore, the value of $f(2)=7$

Comment

**Step 2** of 5

**b)**

The set of all possible dependent value of the outputs of a function is called the Range.

The set of all possible inputs to the function is called its Domain.

From the given values n, and f(n), the Range and Domain are as follows:

• Range $R=\{6,7\}$

• Domain $D=\{1,2,3,4,5\}$

Comment

**Step 3** of 5

**c)**

The value in the function $g(i,j)$, where *i* is the row label and *j* is the column label.

Then the value at the row labeled *2* and the column labeled *10* in the table is $\boxed{g(2,10)=6}$

Comment

**Step 4** of 5

**d)**

The Range of the function *g* is $R = \{6 \quad ,8,9,10\}$

The Domain of the function *g*, which is the input values in the table goes into the function $g(i,j)$

From the table $g(i,j)$

$$\begin{aligned} X &= \{1,2,3,4,5\} \\ Y &= \{6,7,8,9,10\} \\ \text{Domain } D &= X \times Y \\ &= \{(1,6),(1,7),(1,8),(1,9),(1,10),(2,6),(2,7),(2,8), \\ &\quad (2,9),(2,10),(3,6),(3,7),(3,8),(3,9),(3,10)(4,6), \\ &\quad (4,7),(4,8),(4,9),(4,10)(5,6),(5,7),(5,8),(5,9),(5,10)\} \end{aligned}$$

Comment

**Step 5** of 5

**e)**

- If f is a function of the form $f(a)=b$, *b* is the output value when the input value is *a*
- The entry at the row labeled *i* and the column labeled *j* in the table is the value of *g (i, j)*.
- The value of the function f at n=4 is 7, which is $f(4)=7$

$$\begin{aligned} \text{The value of function } g(4, f(4)) &= g(4,7) \\ &= 8 \end{aligned}$$

Comment

## Problem

For each part, give a relation that satisfies the condition.

**a.** Reflexive and symmetric but not tra tive

**b.** Reflexive and transitive but not symmetric

**c.** Symmetric and transitive but not reflexive

## Step-by-step solution

**Step 1** of 4

A binary relation $R$ is

Reflexive if for every $x$, $xRx$

Symmetric if for every $x$ and $y$, $xRy$ implies $yRx$

Transitive if for every $x$, $y$, and z, $xRy$ and $yRz$ implies $x$

Comments (1)

**Step 2** of 4

(a)

Re◊exive and symmetric but not transitive

$xRy$ : $x$, $y$ are people and they share at least one biological parent.

• This relation is clearly re◊exive, since everyone has their own parents for parents.

• This relation is symmetric, if person $x$ shares a parent with person $y$, then person $y$ shares that same parent with person $x$.

• However, this relation is not transitive. Assume a person $m$ who shares exactly one parent (a mother) with person $g$ and exactly one parent with person($s$) (a father). $g$'s father is not $s$'s father, and $g$'s mother is not $s$'s mother. Hence, while $gRm$ and $mRs$ are true, $gRs$ is false.

Comment

**Step 3** of 4

(b)

Re◊exive and transitive but not symmetric

$xRy$ : $x, y \in \mathbb{N}$ and $x - y \leq 0$.

• This relation is Re◊exive, because $x$-$x = 0$.

• This is transitive, because if $xRy$ then $x \geq y$ and if $yRz$ then $y \geq z$, thus $xRx$ because $x \geq y \geq z$.

• However, this relation is not symmetric, because $5 - 3 \geq 0$, but $3 - 5 < 0$.

Comments (1)

**Step 4** of 4

(c)

Symmetric and transitive but not re◊exive

$xRy$ : $x, y \in \mathbb{Z}$ and $i * j > 0$

• This relation is Symmetric, because multiplication is symmetric.

• It is transitive, as well, since if $xRy$ then neither $x$ nor $y$ is zero and if $yRz$ , then neither $y$ nor $z$ is zero. Thus $xRz$ because neither $x$ nor $z$ is zero.

• However, this relation is not symmetric, because $0 * 0 = 0$

## Problem

Consider the undirected graph G=(V,E) where V , the set of nodes, is {1, 2, 3, 4} and E, the set of edges, is {{1, 2}, {2, 3}, {1, 3}, {2, 4}, {1, 4}}. Draw the graph G. What are the degrees of each node? Indicate a path from node 3 to node 4 on your drawing of G.

## Step-by-step solution

**Step 1** of 2

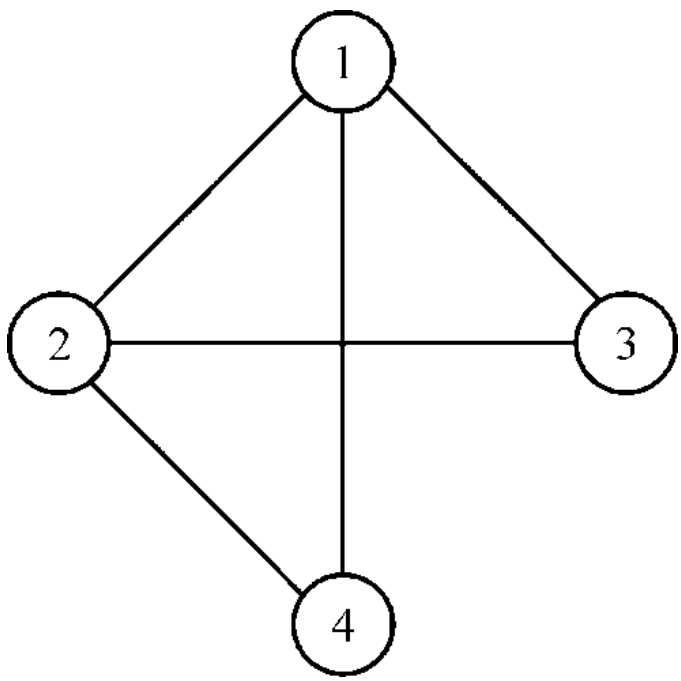
An undirected graph is a set of vertices (nodes) and edges where an edge connects a pair of vertices and it has no orientation.

Consider the undirected graph $G=(V,E)$

Where, *V* is the set of nodes, $\{1,2,3,4\}$

And, *E* is the set of edges, $\{\{1,2\},\{2,3\},\{1,3\},\{2,4\},\{1,4\}\}$.

The graph *G* is as shown below:




**Degree of nodes:**

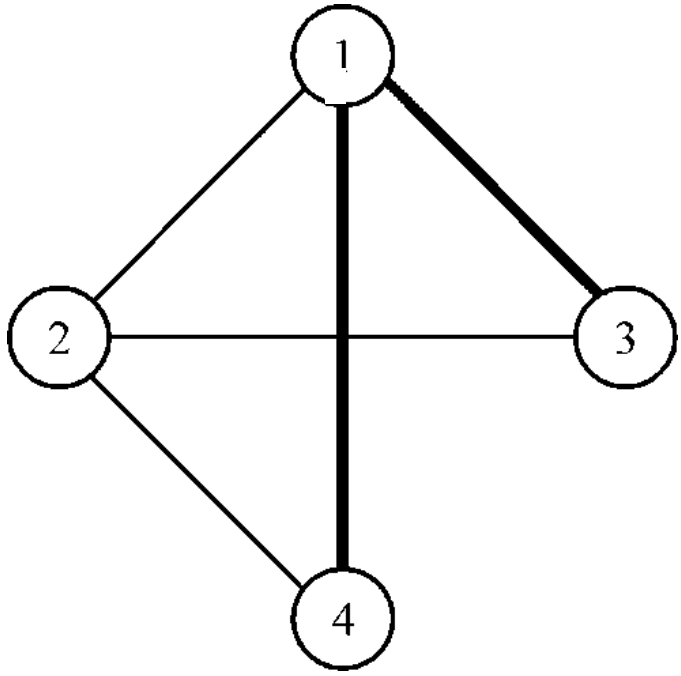
The degree of a node is the number of edges at that particular node. The degrees of each node of graph *G* are as shown in the table below:

| Node | Degree |
|---|---|
| 1 | 3 |
| 2 | 3 |
| 3 | 2 |
| 4 | 2 |

Comment

**Path from node 3 to node 4:**

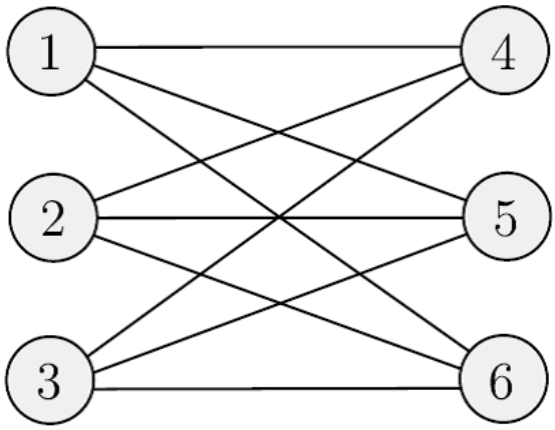
The path from node 3 to node 4 on the graph *G* is as shown below:




Comment

## Problem

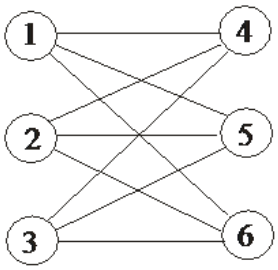
Write a formal description of the following graph.




## Step-by-step solution

**Step 1** of 3

Given graph:




Comment

**Step 2** of 3

Formal description of the graph is

({1, 2, 3, 4, 5, 6}, {(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)})

It shows a set of nodes and a set of edges.

Comment

**Step 3** of 3

In a graph G, we say G = (V, E) where V is the set of nodes and E is the set of edges.

$V = \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}$

$E = \{\{1,\ 4\},\ \{1,\ 5\},\ \{1,\ 6\}, \{2,\ 4\},\ \{2,\ 5\},\ \{2,\ 6\}, \{3,\ 4\},\ \{3,\ 5\},\ \{3,\ 6\}\}$

Comment

## Problem

Find the error in the following proof that 2 = 1. Consider the equation a = b. Multiply both sides by a to obtain $a^2$ = ab. Subtract $b^2$ from both sides to get $a^2$–$b^2$ = ab–$b^2$. Now factor each side, (a+b)(a–b) = b(a–b), and divide each side by (a–b) to get a+b = b. Finally, let a and b equal 1, which shows that 2 = 1.

## Step-by-step solution

**Step 1** of 2

**To prove:** 2 = 1

**Given proof is:**

• Consider the equation $a=b$

• Multiply both sides by a to obtain $a^2=ab$

• Subtract $b^2$ from both sides to get $a^2-b^2=ab-b^2$

• Now factor each side, $(a+b)(a-b)=b(a-b)$

• Divide each side by $(a-b)$, to get $a+b=b$

• Let *a* and *b* equal 1, which shows that 2 = 1.

Comment

**Step 2** of 2

The fallacy lies in the step which involves division by $(a-b)$, which is zero. Since *a* is equal to *b* as stated in the problem statement, *a* – *b* gives zero. Since division by zero is undefined, the argument cannot be considered as valid.

Comment

## Problem

Let S(n) = 1 + 2 + · · · + n be the sum of the first n natural numbers and let C(n) = $1^3 + 2^3 + \cdots + n^3$ be the sum of the first n cubes. Prove the following equalities by induction on n, to arrive at the curious conclusion that C(n) = $S^2$(n) for every n.

**a.** $S(n) = \frac{1}{2}n(n+1).$

**b.** $C(n) = \frac{1}{4}(n^4 + 2n^3 + n^2) = \frac{1}{4}n^2(n+1)^2.$

## Step-by-step solution

**Step 1** of 3

a) The sum of the first n natural numbers $(S_n = 1+2+3+...+n)$ is given by:

$$S_n = \frac{1}{2}n(n+1).$$

Here, induction method is used to prove the above equality. Let's write $S_n = 1+2+3+...+n$ in shorter form like:

$$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1).$$

• For n=1, it is true that

$$1 = \frac{1}{2}1(1+1).$$

• For n=2, it is true that

$$1+2 = \frac{1}{2}2(2+1).$$

• In the same way it is true for n

$$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1).$$

So, finally it has to prove that for n+1 or $S_{n+1} = \sum_{i=1}^{n+1} i = \frac{1}{2}(n+1)(n+2)$.

• If n+1 is added to each side of the next identity:

$$\begin{aligned} S_{n+1} &= 1+2+...+n+n+1 \\ &= S_n + n + 1 \\ &= \frac{n(n+1)}{2} + n + 1 \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

• Which is equivalent to: $\sum_{i=1}^{n+1} i = \frac{1}{2}n(n+1) + \frac{2(n+1)}{2}$ or $S_{n+1} = \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$

So, the above equality proves that the given equality for the sum of n natural or $S_n = \frac{1}{2}n(n+1)$ is also true for $(n+1)$. Hence the given equality is correct.

Comment

**Step 2** of 3

**b)** The sum of the cube of the first n natural numbers or $C_n = 1^3 + 2^3 + ... + n^3$ is given by:

$$C_n = \frac{1}{4}\left(n^4 + 2n^3 + n^2\right) = \frac{1}{4}n^2\left(n+1\right)^2$$

The above equality can be proved by induction method. Let's write $C_n = \frac{1}{4}n^2\left(n+1\right)^2$ in shorter form like:

$$\sum_{i=1}^{k=n} i^3 = 1^3 + 2^3 + ... + n^3 = \frac{1}{4}n^2\left(n+1\right)^2$$

• For n=1, it is true that:

$1^3 = \frac{1}{4}1^2\left(1+1\right)^2$ or 1=1.

• In the same way it is true for n:

$$\sum_{i=1}^{k=n} i^3 = 1^3 + 2^3 + ... + n^3 = \frac{1}{4}n^2\left(n+1\right)^2.$$

So, finally it has to prove that for n+1 or $C_n = 1^3 + 2^3 + ... + n^3 + \left(n+1\right)^3 = \frac{\left(n+1\right)^2\left(n+2\right)^2}{4}$

$$\begin{aligned} C_{n+1} &= 1^3 + 2^3 + 3^3 + ... + n^3 + (n+1)^3 \\ &= \frac{1}{4}n^2(n+1)^2 + (n+1)^3 \\ &= (n+1)^2\left(\frac{n^2+4n+4}{4}\right) \\ &= \frac{(n+1)^2(n+2)^2}{4} \end{aligned}$$

Therefore,

$$C_{n+1} = \frac{\left(n+1\right)^2\left(n+2\right)^2}{4}$$

So, the above equality proves that the given equality for the sum of the cube of n natural numbers is also true for n+1. Hence, the given equality is correct.

Comment

**Step 3** of 3

From the above explanation:

$$C_n = \frac{1}{4}\left(n^4 + 2n^3 + n^2\right) = \frac{1}{4}n^2\left(n+1\right)^2 = \left[\frac{n\left(n+1\right)}{2}\right]^2 = \left(S_n\right)^2$$

$$C_n = \left(S_n\right)^2.$$

It is concluded that "the sum of the cube of the first n natural number is equal to the square of the sum of first n natural number".

Comment

## Problem

Find the error in the following proof that all horses are the same color.

CLAIM: In any set of h horses, all horses are the same color.

PROOF: By induction on h.

***Basis:*** For h = 1. In any set containing just one horse, all horses clearly are the same color.

***Induction step:*** For k ≥ 1, assume that the claim is true for h = k and prove that it is true for h = k+1. Take any set H of k+1 horses. We show that all the horses in this set are the same color. Remove one horse from this set to obtain the set $H^1$ with just k horses. By the induction hypothesis, all the horses in $H^1$ are the same color. Now replace the removed horse and remove a different one to obtain the set $H^2$. By the same argument, all the horses in $H^2$ are the same color. Therefore, all the horses in H must be the same color, and the proof is complete.

## Step-by-step solution

**Step 1** of 2

The error has occurred in the last sentence. There are two statements "all horses in $H_1$ are the same or identical in color" And "all horses in $H_2$ are identical or the same color" for the choice of $H_1$ and $H_2$ which has taken.

Comment

**Step 2** of 2

The induction is valid only for value greater than 2 but fails for 2.

• It indicates that all the horses present in $H = H_1 \cup H_2$ are identical (that is, the same color) if $H = H_1 \cap H_2$ is not empty which returns always true result except when exactly two horses $(h = 2)$ contained in $H$. Then exactly one horse out of the two horses will be consist in each $H_1$ and $H_2$.

• Certainly the single horse in $H_1$ and only the single horse in $H_2$ has its peculiar color. Therefore, it has no meaning to conclude that the horses have the same color.

• So the base case should be for value 2 not for 1. Since the base case is false, the above proof is wrong.

Comments (2)

## Problem

Show that every graph with two or more nodes contains two nodes that have equal degrees.

## Step-by-step solution

**Step 1** of 1

Let us consider a graph G which has at least one edge and without having any loops or cycles in the graph. In graph G we would prove that there are at least two nodes with degree 1 i.e. with equal degree. This conclusion would be taken forward for a graph with 2 or more nodes.

In a graph G we must get a node, say $V_1$ at which only one edge is incident, i.e. $\text{degree}(\text{V1})=1$. Let $E_1$ be this edge which is incident at $V_1$. Since $G$ has no cycles other end of $E_1$ is not $V_1$. Let it be $V_2$. If there exists no other edges which is incident at $V_2$ then $\text{degree}(\text{V2})=1$.

Otherwise let $E_2$ be the edge which is incident at $V_2$. Arguing similar way and proceeding in this way we get a node $V_k$ having degree 1 and is equal to degree of $V_1$. Thus we get that graph G has at least two nodes of degree 1. Thus two nodes have the same degrees in a graph with two or more nodes.

Comment

## Problem

**Ramsey's theorem.** Let G be a graph. A ***clique*** in G is a subgraph in which every two nodes are connected by an edge. An ***anti-clique***, also called an ***independent set***, is a subgraph in which every two nodes are not connected by an edge. Show that every graph with n nodes contains either a clique or an anti-clique with at least 1/2 $\log^2$ n nodes.

## Step-by-step solution

**Step 1** of 2

Consider the graph *G*. A clique is a subgraph of *G* in which every pair of vertices are connected. An anti-clique is a subgraph in which in which every pair of vertices are not connected.

Comment

**Step 2** of 2

In order to show that every graph with *n* vertices contains either clique or anti-clique with at least $\frac{1}{2}\log_2 n$ vertices, create two piles *A* and *B* to store the vertices of a graph. Here, the pile *A* contains the vertices of a clique whereas the pile *B* contains the vertices of an anti-clique.

Procedure to identify a clique or an anti-clique is as follows:

• Take each vertex *v* of the graph *G*.

• If the degree of the vertex is greater than one half of the remaining vertices then add the vertex to pile *A*. Otherwise, add the vertex to the pile *B*.

• Discard all vertices to which *v* is not connected if it was added to the pile *A*.

• Discard all vertices to which *v* is connected if it was added to the pile *B*.

• Continue this procedure until no vertices left.

Consider the whole procedure as a step. For each step, at most half of the vertices are discarded. Thus, at least $\log_2 n$ steps occur before completion of the process. Each step adds a vertex to one of the piles. Thus, one of the piles contains at least $\frac{1}{2}\log_2 n$ vertices.

Therefore, it is proved that that every graph with *n* vertices contains either clique or anti-clique with at least $\frac{1}{2}\log_2 n$ vertices.

Comment

## Problem

Use Theorem 0.25 to derive a formula for calculating the size of the monthly payment for amortgage in terms of the principal P, the interest rate I, and the number of payments t. Assume that after t payments have been made, the loan amount is reduced to 0. Use the formula to calculate the dollar amount of each monthly payment for a 30-year mortgage with 360 monthly payments on an initial loan amount of $100,000 with a 5% annual interest rate.

## Step-by-step solution

**Step 1** of 3

Given formula related to loan is

$$P_t = PM^t - Y\left(\frac{M^t - 1}{M - 1}\right)$$

where,

$P_t$ is the amount of loan outstanding after the $t^{\text{th}}$ month.

$P$ is the principal (original loan amount).

$Y$ is the monthly payment.

$t$ is the number of months in which loan is repaid.

$I$ is the yearly interest rate.

$M$ is the monthly multiplier $(M = 1 + I/12)$.

Now we have to derive the formula for calculating the size of the monthly payments for a mortgage in terms of the principal $P$, interest rate $I$, and the number of payments $t$.

Comment

**Step 2** of 3

In order to derive the formula we have to get $Y$ (monthly payment) on left hand side and remaining terms to right hand side.

$$P_t = PM^t - Y\left(\frac{M^t - 1}{M - 1}\right)$$

$$Y\left(\frac{M^t - 1}{M - 1}\right) = PM^t - P_t$$

$$Y = \left(\frac{M - 1}{M^t - 1}\right)\left(PM^t - P_t\right)$$

The formula required for calculation is

$$Y = \left(\frac{M - 1}{M^t - 1}\right)\left(PM^t - P_t\right)$$

Comment

**Step 3** of 3

Given data is

$$P_t = \$0$$
$$P = \$100,000$$
$$t = 360 \text{ months}$$
$$I = 5\% = \frac{5}{100} = 0.05$$
$$M = 1 + I$$
$$= 1 + \frac{0.05}{12} \quad = 1.0042(\text{approx})$$

We get

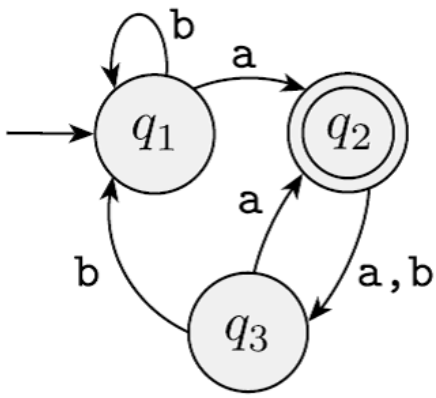
$$Y = \left(\frac{M-1}{M^t - 1}\right)\left(PM^t - P_t\right)$$
$$= \left(\frac{1.00417-1}{1.00417^{360}-1}\right)\left(100000 \times 1.00417^{360} - 0\right)$$
$$= \left(\frac{0.00417}{3.47309}\right)\left(100000 \times 4.47309\right)$$
$$= 0.0012 \times 447309$$
$$\approx 536.7708$$

**Therefore, the monthly payment is $536.78.**

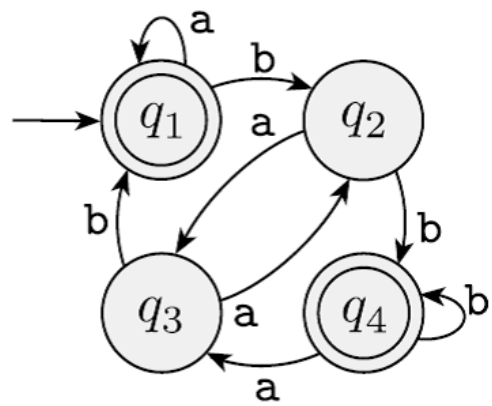
Comment

## Problem

The following are the state diagrams of two DFAs,$M_1$ and$M_2$. Answer the following questions about each of these machines.




$M_1$




$M_2$

**a.** What is the start state?

**b.** What is the set of accept states?

**c.** What sequence of states does the machine go through on input aabb?

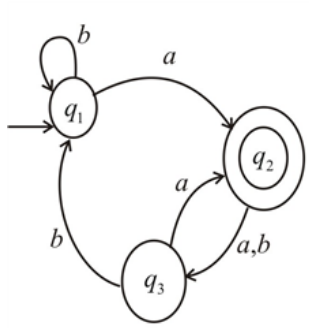
**d.** Does the machine accept the string aabb?

**e.** Does the machine accept the string "?

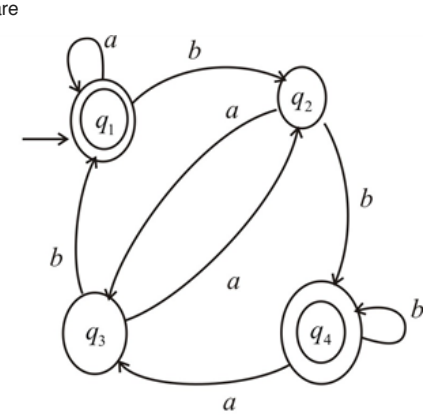
## Step-by-step solution

**Step 1** of 7

State diagrams for two DFAs, $M_1$ and $M_2$ are




$M_1$




$M_2$

Comment

**Step 2** of 7

(a) Start state is indicated by the arrow pointing towards it.

• For the DFA (Deterministic Finite Automata) $M_1$, an arrow is pointing towards the state $q_1$. So $q_1$ is the start state.

• For the DFA $M_2$, arrow is pointing towards the state $q_1$. So $q_1$ is the start state.

Comment

**Step 3** of 7

(b) The accept state is the one which is identified by a double circle.

• In state diagram of $M_1$, the state $q_2$ has double circle. So $\{q_2\}$ is the accept state.

• In the sate diagram of $M_2$, the states $q_1$ and $q_4$ have double circles. So the set $\{q_1, q_4\}$ are the accept states.

Comment

**Step 4** of 7

(c)

• When we given the string *aabb* as input to the machine $M_1$ then the following transitions will takes place

(1) Start in state $q_1$

(2) Read *a*, follow transition from $q_1$ to $q_2$

(3) Read *a,* follow transition from $q_2$ to $q_3$.

(4) Read *b*, follow transition from $q_3$ to $q_1$.

(5) Read *b*, follow transition from $q_1$ to $q_1$

From (1), (2), (3), (4), (5)

So the machine $M_1$ will go through following sequence of states on input *aabb*. $q_1, q_2, q_3, q_1, q_1$

• When we give the input *aabb* to machine $M_2$ then the following transitions will takes place

(1) Start in state $q_1$

(2) Read *a*, follow transition from $q_1$ to $q_1$

(3) Read *b*, follow transition from $q_1$ to $q_2$

(5) Read *b*, follow transition from $q_2$ to $q_4$

From (1), (2), (3), (4), (5)

The machine $M_2$ will go through the following sequence of states on input *aabb*

$q_1, q_1, q_1, q_2, q_4$

Comment

**Step 5** of 7

(d)

• Machine $M_1$ will go through following sequence of sates on input *aabb*

1. Start in state $q_1$

2. Read *a*, follow transition from $q_1$ to $q_2$

3. Read *a*, follow transition from $q_2$ to $q_3$

4. Read *b*, follow transition from $q_3$ to $q_1$

5. Read *b*, follow transition from $q_1$ to $q_1$

On reading the input *aabb*, $M_1$ finally entered into state $q_1$, which is not an accept state.

So $M_1$ reject the input *aabb*.

• Machine $M_2$ will go through the following sequence of sates on input *aabb*

1. Start in state $q_1$

2. Read *a*, follow transition from $q_1$ to $q_1$

3. Read *a*, follow transition from $q_1$ to $q_1$

4. Read *b*, follow transition from $q_1$ to $q_2$

5. Read *b*, follow transition from $q_2$ to $q_4$

On reading the input *aabb*, $M_2$ finally entered into state $q_4$.

$q_4$ accept state of $M_2$.

Hence $M_2$ accept the input *aabb*.

Comment

**Step 6** of 7

(e)

• On giving the input $\in$(empty string) to machine $M_1$, $M_1$ always in the start state $q_1$.

But $\{q_1\}$ in not the accept sate of $M_1$.

Thus $M_1$ does not accept the string $\in$

•

Comment

**Step 7** of 7

On giving the input $\in$(empty string) to the machine $M_2$, $M_2$ always in the start state $q_1$.

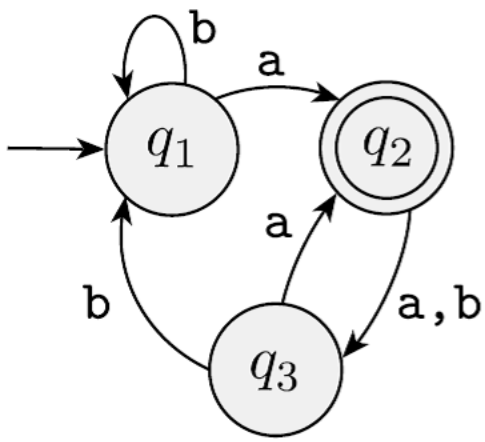
But $\{q_1\}$ is also accept state of $M_2$.

Thus $M_2$ accept the string $\in$.

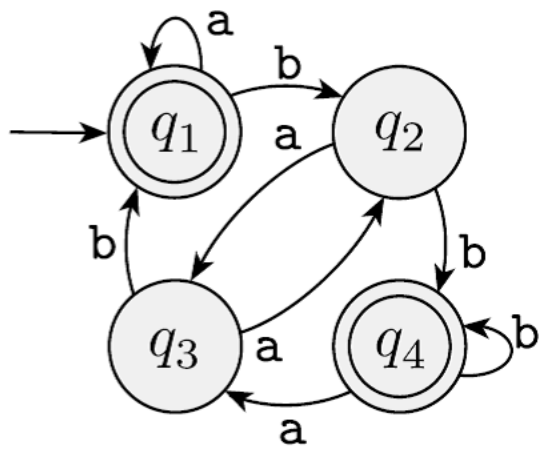
Comment

**Problem**

Give the formal description of the machinesM1 and M2 pictured in Exercise 1.1.




$M_1$

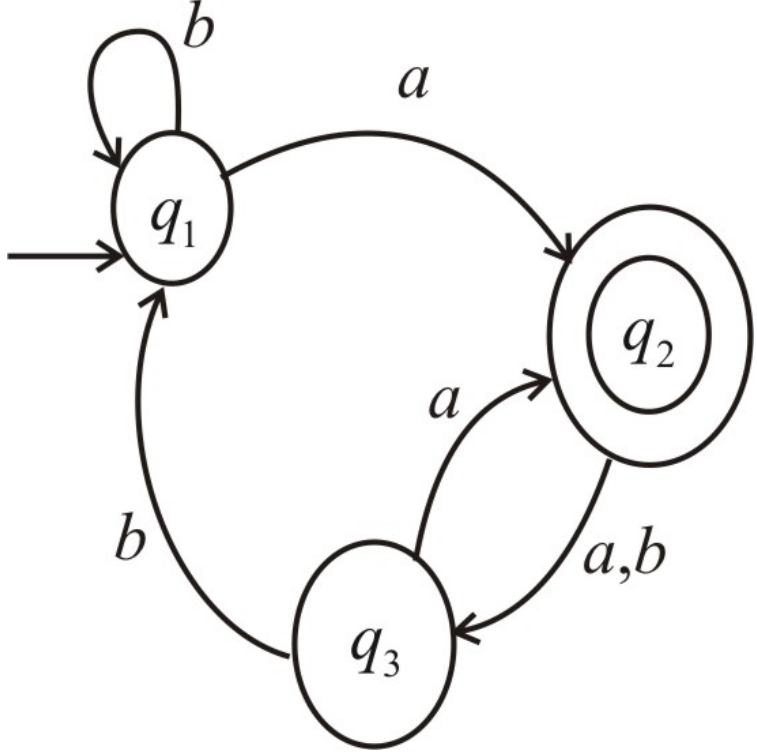



$M_2$

**Step-by-step solution**

**Step 1** of 4

Given state diagram for the machine $M_1$ is




Comment

**Step 2** of 4

Formal definition of a finite automata is

A finite automaton is a 5-tupel $(Q,\Sigma,\delta,q_0,F)$

Where

1. $Q$ is a finite set called states
2. $\Sigma$ is a finite set called alphabet
3. $\delta : Q\times\Sigma \to Q$ is the transition function.
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept states.

Now we can describe $M_1$ formally by writing $M_1 = (Q,\Sigma,\delta,q_1,F)$, where

1. $Q = \{q_1,q_2,q_3\}$
2. $\Sigma = \{a,b\}$
3. $\delta$ is described as

| | $a$ | $b$ |
|---|---|---|
| $q_1$ | $q_2$ | $q_1$ |
| $q_2$ | $q_3$ | $q_3$ |
| $q_3$ | $q_2$ | $q_3$ |

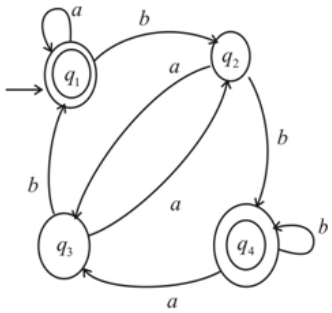
4. $q_1$ is the start state

5. Set of accept states $F = \{q_2\}$.

Comments (3)

**Step 3** of 4

Given state diagram for the machine $M_2$ is




$M_2$

Comment

**Step 4** of 4

Now we can describe $M_2$ formally by writing

$M_2 = (Q, \Sigma, \delta, q_1, F)$, where

1. Set of states $Q = \{q_1, q_2, q_3, q_4\}$

2. Set of alphabet $\Sigma = \{a, b\}$

3. $\delta$ is described as

| | $a$ | $b$ |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_4$ |
| $q_3$ | $q_2$ | $q_1$ |
| $q_4$ | $q_3$ | $q_4$ |

4. $q_1$ is the start state

5. Set of accept states $F = \{q_1, q_4\}$

Comments (4)

## Problem

The formal description of a DFA M is {q1, q2, q3, q4, q5}, {u, d}, $\delta$, $q_3$, q3, {q3} ), where _ is given by the following table. Give the state diagram of this machine.

| | u | d |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_1$ | $q_3$ |
| $q_3$ | $q_2$ | $q_4$ |
| $q_4$ | $q_3$ | $q_5$ |
| $q_5$ | $q_4$ | $q_5$ |

## Step-by-step solution

**Step 1** of 1

Given formal description of DFA *M* is

$M = (Q, \Sigma, \delta, q_3, F)$

$Q =$ Set of states $= \{q_1, q_2, q_3, q_4, q_5\}$

$\Sigma =$ Set of alphabet $= \{u, d\}$

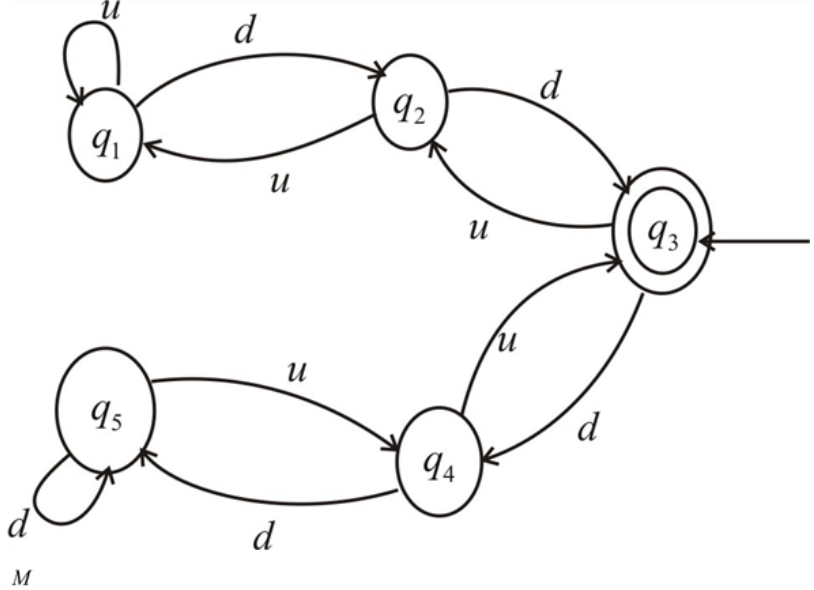
$\delta =$ The transition function is described as

| | $u$ | $d$ |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_1$ | $q_3$ |
| $q_3$ | $q_2$ | $q_4$ |
| $q_4$ | $q_3$ | $q_4$ |
| $q_5$ | $q_4$ | $q_5$ |

Start state $= \{q_3\}$ indicated with an arrow

Set of accept states Final state $= \{q_3\}$ indicated by double circle

Now we will construct state diagram by using the above details.




So this is the state diagram for the given description of machine *M*.

Comments (2)

## Problem

Each of the following languages is the intersection of two simpler languages. In each part, construct DFAs for the simpler languages, then combine them using the construction discussed in footnote 3 (page 46) to give the state diagram of a DFA for the language given. In all parts, ∑ = {a, b}.

**a.** {wl w has at least three a's and at least two b's}

A**b.** {wl w has exactly two a's and at least two b's}

**c.** {wl w has an even number of a's and one or two b's}

A**d.** {wl w has an even number of a's and each a is followed by at least one b}

**e.** {wl w starts with an a and has at most one b}

**f.** {wl w has an odd number of a's and ends with a b}

**g.** {wl w has even length and an odd number of a's}

## Step-by-step solution

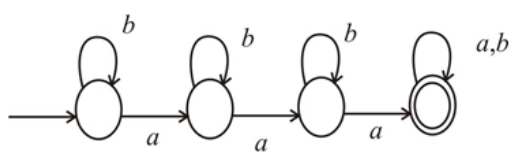
**Step 1** of 18

a.

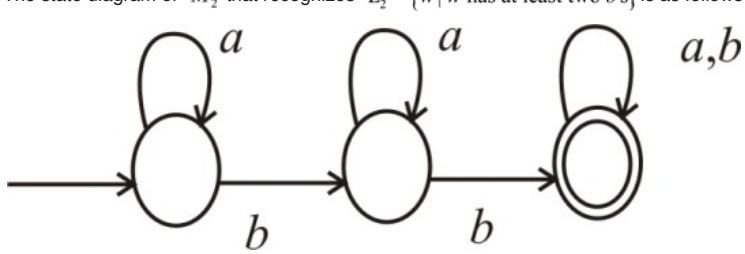
Consider the Language $L = \{w \mid w \text{ has at least three } a\text{'s and at least two } b\text{'s}\}$. The language *L* is the intersection of two simpler languages $L_1$ and $L_2$.

Now $L_1 = \{w \mid w \text{ has at least three } a\text{'s}\}$ and $L_2 = \{w \mid w \text{ has at least two } b\text{'s}\}$. Let *M* be the DFA (Deterministic Finite automata) that recognizes *L*. Let $M_1$ and $M_2$ be the DFAs that recognizes $L_1$ and $L_2$.

The state diagram of $M_1$ that recognizes $L_1 = \{w \mid w \text{ has at least three } a\text{'s}\}$ is as follows:



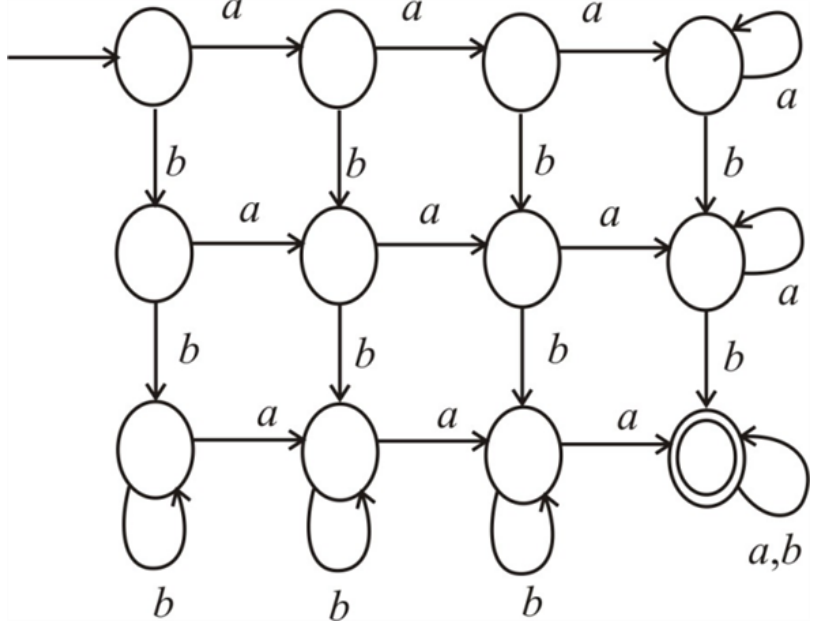

The state diagram of $M_2$ that recognizes $L_2 = \{w \mid w \text{ has at least two } b\text{'s}\}$ is as follows:




Comments (1)

**Step 2** of 18

The machine *M* will accept the input if and only if both $M_1$ and $M_2$ accept it. Because, the language *L* is the intersection of the languages $L_1$ and $L_2$. Therefore, the state diagram of *M* that recognizes the Language $L$ is as follows:




Comment

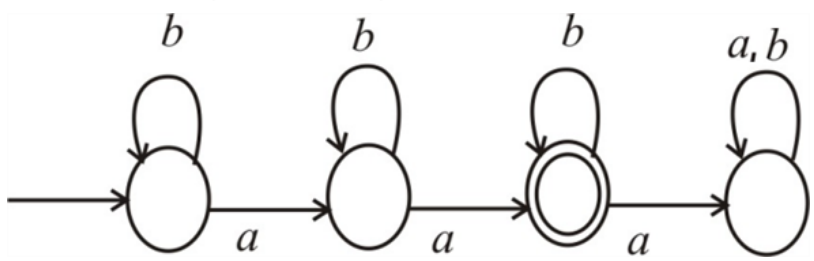
**Step 3** of 18

b.

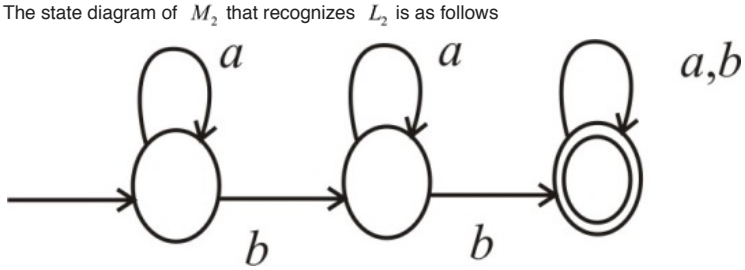
Consider the language $L = \{ w \mid w$ has at exactly two *a*'s and at least two *b*'s$\}$. The language *L* is the intersection of two simpler languages say $L_1$ and $L_2$.

Now $L_1 = \{ w \mid w$ has exactly two *a*'s$\}$ and $L_2 = \{ w \mid w$ has at least two *b*'s$\}$. Let *M* be the DFA(Deterministic Finite automata) that recognizes *L* and $M_1$ and $M_2$ be the DFAs that recognizes $L_1$ and $L_2$.

The state diagram of $M_1$ that recognizes $L_1$ is as follows



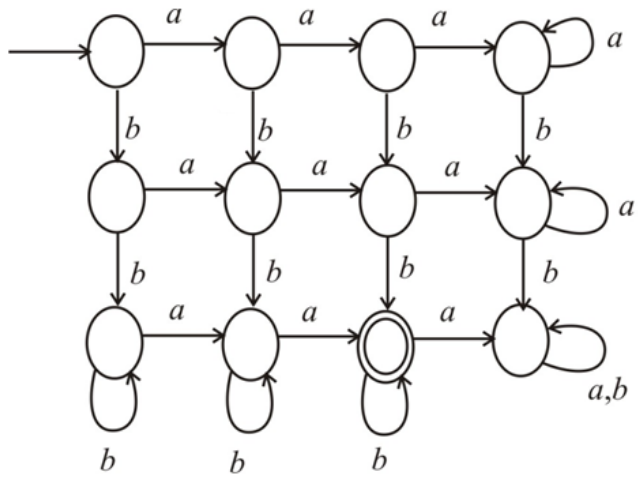

The state diagram of $M_2$ that recognizes $L_2$ is as follows




Comments (1)

**Step 4** of 18

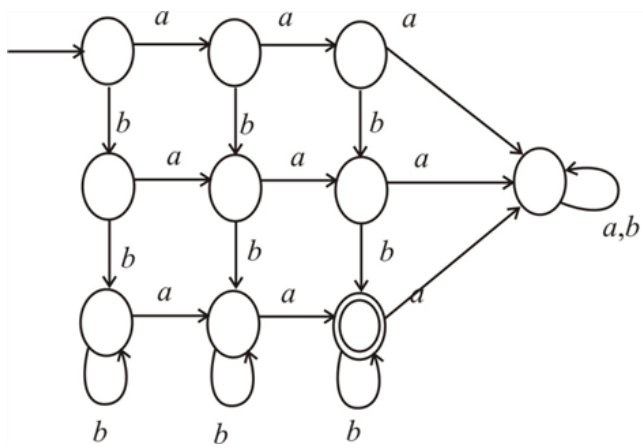
The machine *M* will accept the input if and only if both $M_1$ and $M_2$ accept it. Because, the language *L* is the intersection of $L_1$ and $L_2$. The state diagram of *M* that recognizes *L* is as follows:




Combine some states to get more simplified form as follows:

Comment

**Step 5** of 18




Comments (1)

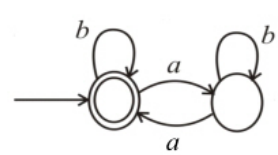
**Step 6** of 18

c.

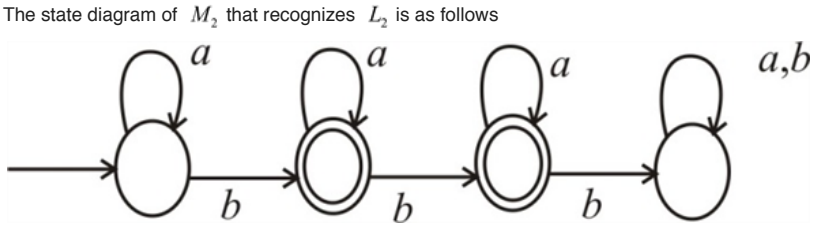
Consider the language $L = \{ w \mid w$ has even number of *a*'s and one or two *b*'s}. The language *L* is the intersection of two simpler languages say $L_1$ and $L_2$.

Now $L_1 = \{ w \mid w$ has even number of *a*'s } and $L_2 = \{ w \mid w$ has one or two *b*'s}. Let *M* be the DFA(Deterministic Finite automata) that recognizes *L* and $M_1$ and $M_2$ be the DFAs that recognizes $L_1$ and $L_2$.

The state diagram of $M_1$ that recognizes $L_1$ is as follows




The state diagram of $M_2$ that recognizes $L_2$ is as follows

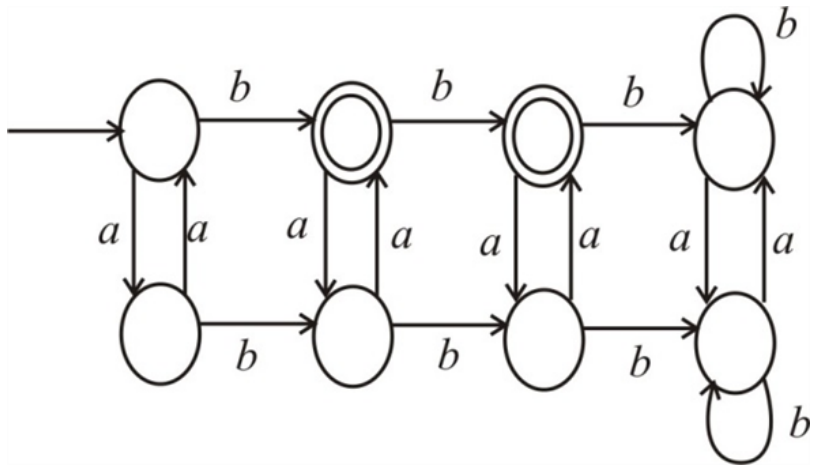



Comment

**Step 7** of 18

The machine *M* will accept the input if and only if both $M_1$ and $M_2$ accept it. Because language *L* is the intersection of $L_1$ and $L_2$.

The state diagram of *M* which recognizes the language *L* is as follows




Comments (3)

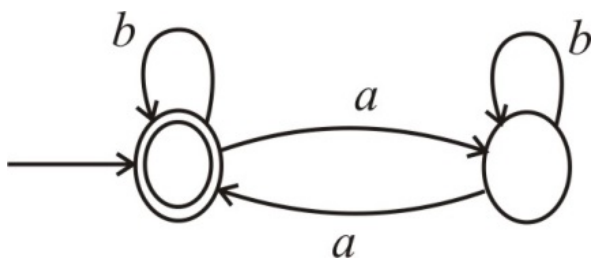
**Step 8** of 18

d.

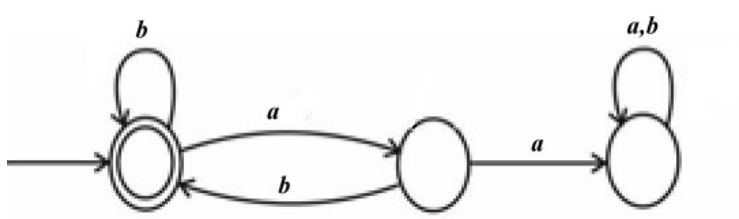
Consider the language $L = \{ w \mid w$ has an even number of *a*'s and each a is a followed by at least on *b*}. The language *L* is the intersection of two simpler languages say $L_1$ and $L_2$.

Now $L_1$={ $w \mid w$ has an even number of *a*'s } and $L_2$={ $w \mid w$ has each a is a followed by at least on *b* }. Let *M* be the DFA(Deterministic Finite automata) that recognizes *L* and $M_1$ and $M_2$ be the DFAs that recognizes $L_1$ and $L_2$.

The state diagram of $M_1$ that recognizes $L_1$ is as follows where $L_1 = \{ w \mid w$ has an even number of *a*'s}.




The state diagram of $M_2$ that recognizes $L_2$ is as follows where $L_2 = \{ w \mid w$ has each *a* is followed by at least one *b*}.




Comment

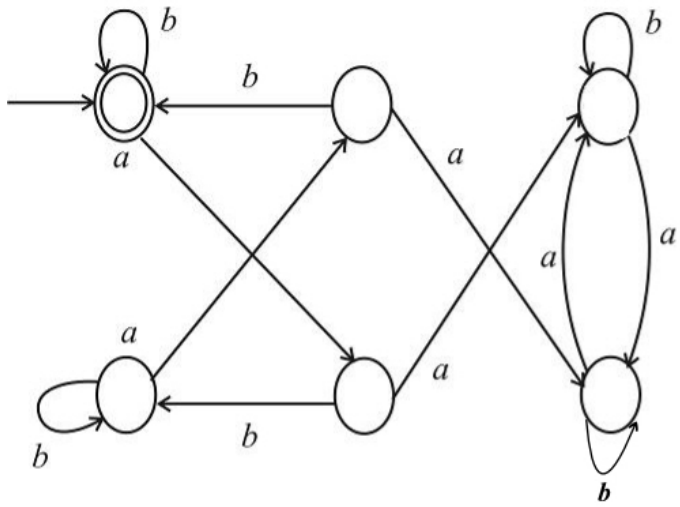
**Step 9** of 18

The machine *M* will accept the input if and only if both $M_1$ and $M_2$ accept it. Because, the language *L* is the intersection of $L_1$ and $L_2$.

The state diagram of *M* that recognizes *L* is as follows:

Comment

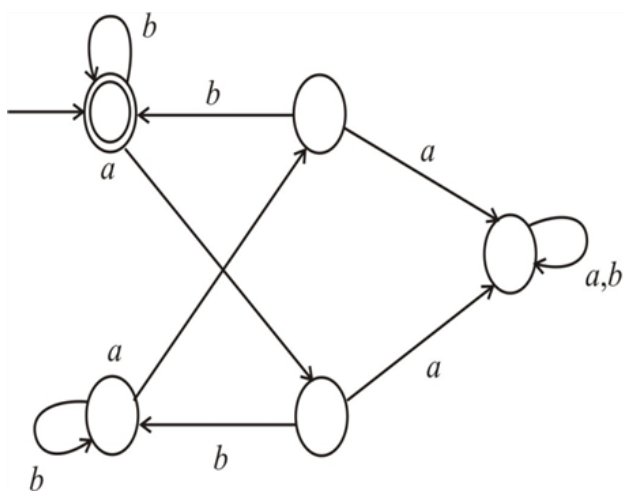
**Step 10** of 18




Comments (4)

**Step 11** of 18

Combine some states to get more simplified form as follows:



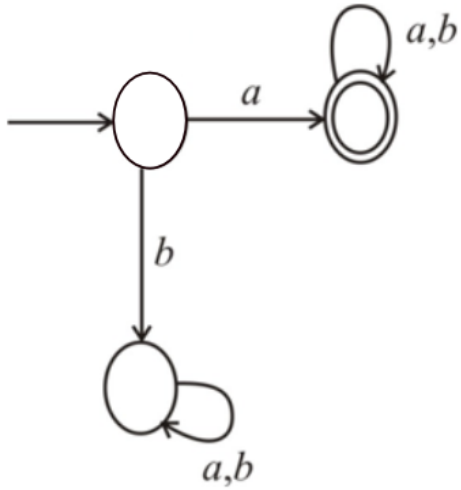

Comments (1)

**Step 12** of 18

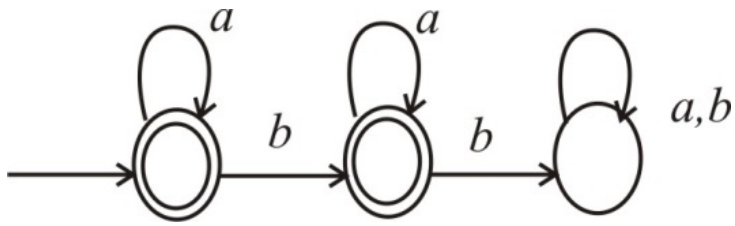
e.

Consider the language $L = \{ w \mid w$ Start with an *a* and has at most one *b*$\}$. The language *L* is the intersection of two simpler languages say $L_1$ and $L_2$. Now $L_1 = \{ w \mid w$ starts with *a*$\}$and $L_2 = \{ w \mid w$ has at most one *b*$\}$. Let *M* be the DFA(Deterministic Finite automata) that recognizes *L* and $M_1$ and $M_2$ be the DFAs that recognizes $L_1$ and $L_2$.

The state diagram of $M_1$ that recognizes $L_1$ is as follows:



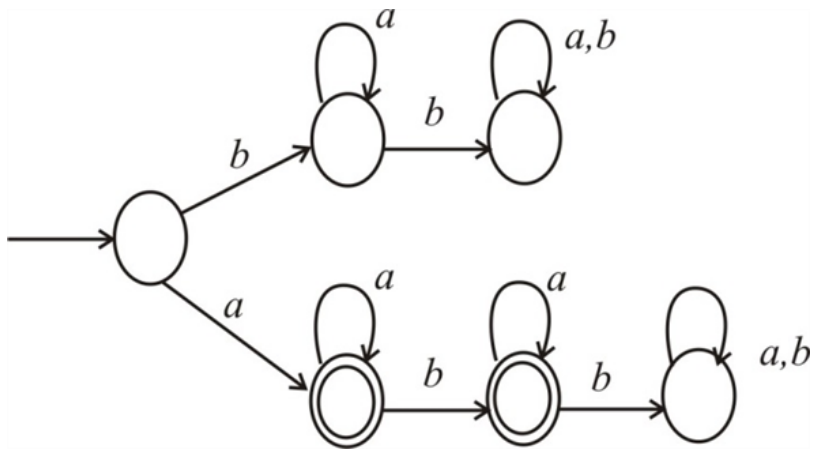

The state diagram of $M_2$ that recognizes $L_2$ is as follows.



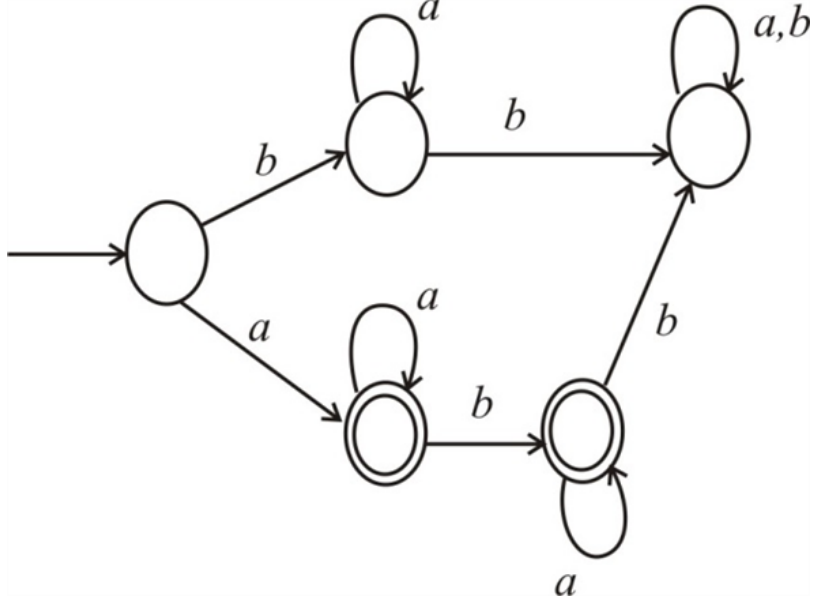

Comment

**Step 13** of 18

The machine *M* will accept the input if and only if both $M_1$ and $M_2$ accept it. Because, the language *L* is the intersection of $L_1$ and $L_2$. The state diagram of *M* that recognizes *L* is as follows.




Combine some states to get more simplified form as follows:




Comments (3)

**Step 14** of 18

f.

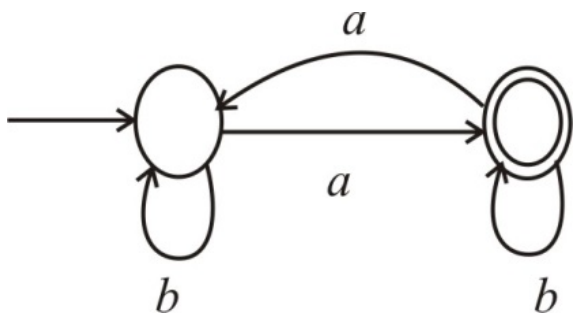
Consider the language $L = \{ w \mid w$ has an odd number of *a*'s and ends with *b*$\}$. The language *L* is the intersection of two simpler languages say $L_1$ and $L_2$. Now $L_1 = \{ w \mid w$ has an odd

Comment

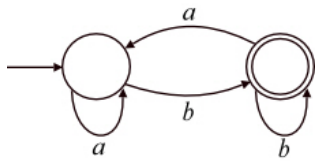
**Step 15** of 18

number of *a*'s$\}$ and $L_2 = \{ w \mid w$ ends with a *b*$\}$. Let *M* be the DFA(Deterministic Finite automata) that recognizes *L* and $M_1$ and $M_2$ be the DFAs that recognizes $L_1$ and $L_2$.

The state diagram of $M_1$ that recognizes $L_1$ is as follows:




The language $L_2$ accepts the strings that ends with a symbol *b*. The state diagram of $M_2$ that recognizes $L_2$ is as follows:

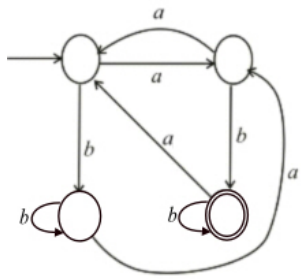



Comment

**Step 16** of 18

The machine *M* will accept the input if and only if both $M_1$ and $M_2$ accept it. Because, the language *L* is the intersection of $L_1$ and $L_2$.

The state diagram of *M* that recognizes *L* is as follows:



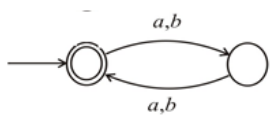

Comments (9)

**Step 17** of 18

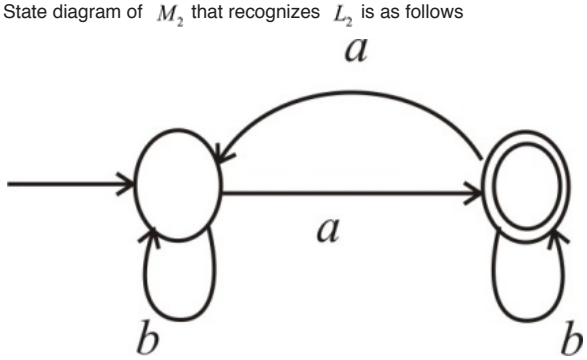
g.

Consider the language $L = \{ w \mid w$ has even length and an odd number of *a*'s $\}$. The language *L* is the intersection of two simpler languages say $L_1$ and $L_2$. Now $L_1 = \{ w \mid w$ has even length $\}$ and $L_2 = \{ w \mid w$ has odd number of *a*'s $\}$. Let *M* be the DFA(Deterministic Finite automata) that recognizes *L* and $M_1$ and $M_2$ be the DFAs that recognizes $L_1$ and $L_2$.

State diagram of $M_1$ that recognizes $L_1$ is as follows




State diagram of $M_2$ that recognizes $L_2$ is as follows

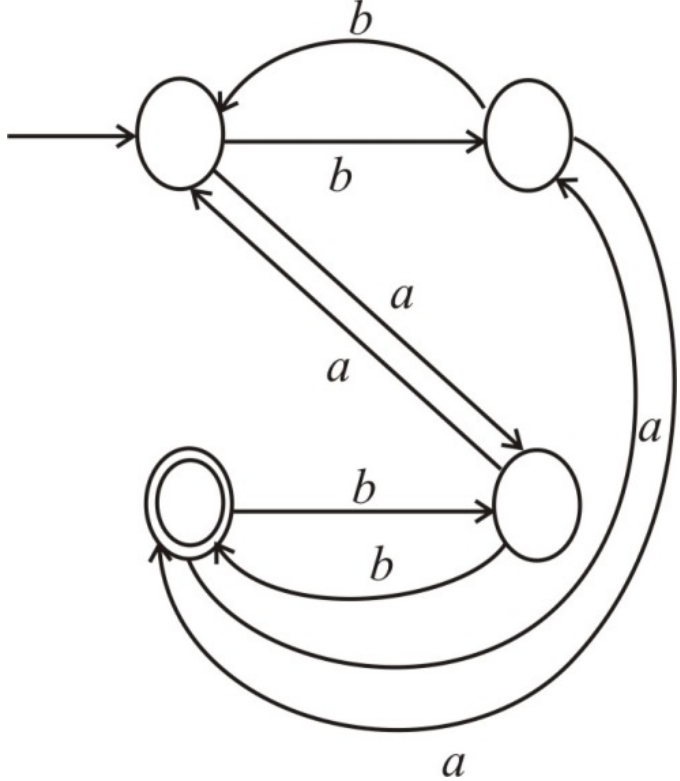



Comment

**Step 18** of 18

The machine *M* will accept the input if and only if both $M_1$ and $M_2$ accept it. Because, the language *L* is the intersection of $L_1$ and $L_2$.

The state diagram of *M* that recognize *L* is as follows:




Comments (2)

## Problem

Each of the following languages is the complement of a simpler language. In each part, construct a DFA for the simpler language, then use it to give the state diagram of a DFA for the language given. In all parts, ∑ = {a, b}.

A**a.** {wl w does not contain the substring ab}

A**b.** {wl w does not contain the substring baba}

**c.** {wl w contains neither the substrings ab nor ba}

**d.** {wl w is any string not in a*b*}

**e.** {wl w is any string not in (ab**+**)*}

**f.** {wl w is any string not in a* ∪ b*}

**g.** {wl w is any string that doesn't contain exactly two a's}

**h.** {wl w is any string except a and b}

## Step-by-step solution

**Step 1** of 16

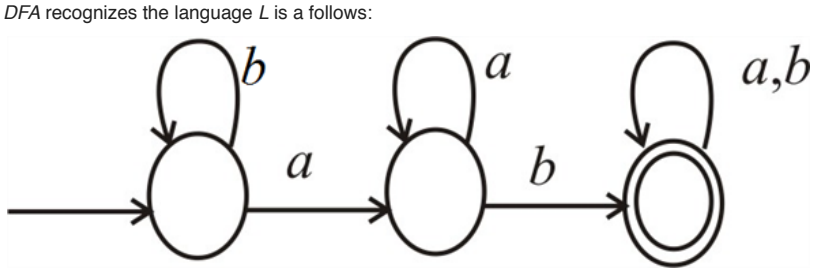
(a) The language is

$\overline{L}$ ={ $w$ | $w$ does not contain the substring *ab*}

$\overline{L}$ is the complement of a simpler language *L*.

Then the simple language is $L$ ={ $w$ | $w$ contain the substring *ab*}

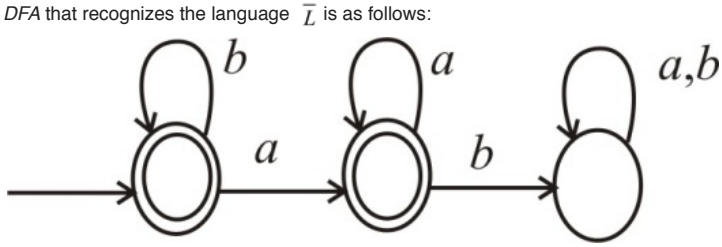
*DFA* recognizes the language *L* is a follows:




Comments (4)

**Step 2** of 16

*DFA* that recognizes the language $\overline{L}$ is as follows:




Comment

**Step 3** of 16

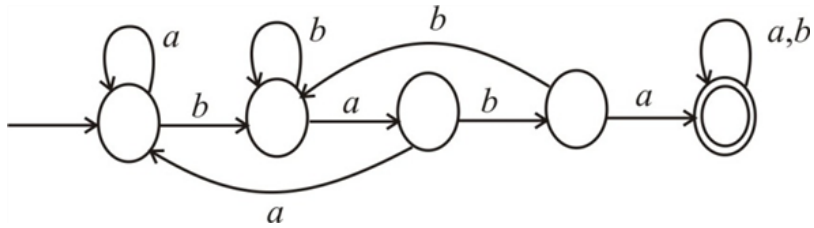
(b) The language is

$\overline{L} = \{ w \mid w$ does not contain the substring *baba*$\}$

$\overline{L}$ is the complement of a simpler language *L*.

Then the simple language is $L = \{ w \mid w$ contain the substring *baba*$\}$

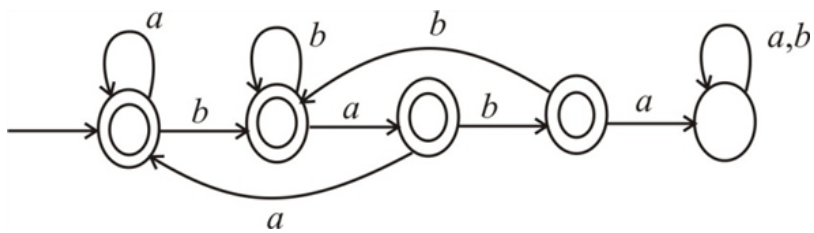
*DFA* that recognizes the language *L* is as follows:




Comment

**Step 4** of 16

*DFA* that recognizes the language *L* is as follows:




Comment

**Step 5** of 16

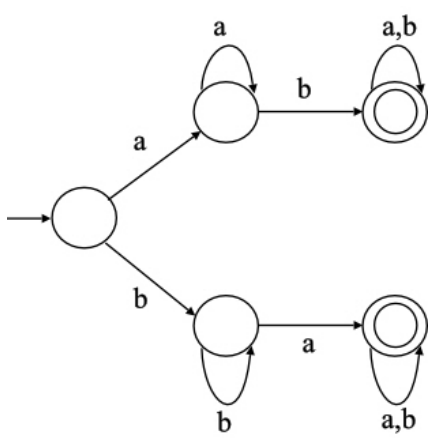
(c) The language is

$\overline{L} = \{ w \mid w$ contains neither the substrings *ab* nor *ba*$\}$

$\overline{L}$ is the complement of a simpler language *L*.

Then the simple language is $L = \{ w \mid w$ contains either the substring *ab* or *ba*$\}$

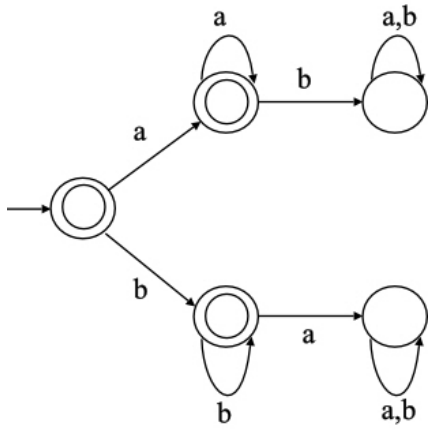
*DFA* that recognizes the language *L* is as follows




*DFA* that recognizes the language $\overline{L}$ is as follows:

Comment

**Step 6** of 16




Comment

**Step 7** of 16

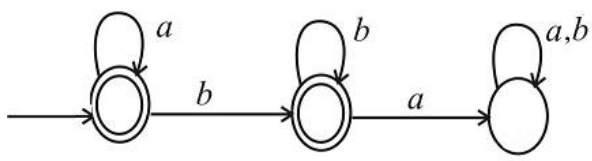
(d) The language is

$\overline{L} = \{ w \mid w$ is any string not in $a^*b^*\}$

$\overline{L}$ is the complement of a simpler language *L*.

Then the simple language is $L = \{ w \mid w$ is any string in $a^*b^*\}$

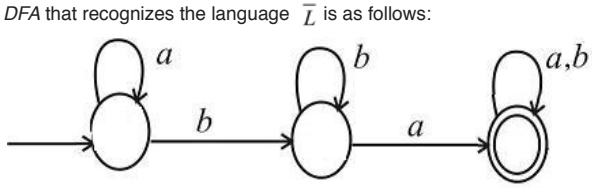
*DFA* that recognizes the language *L* as follows




Comments (7)

**Step 8** of 16

*DFA* that recognizes the language $\overline{L}$ is as follows:




Comments (4)

**Step 9** of 16

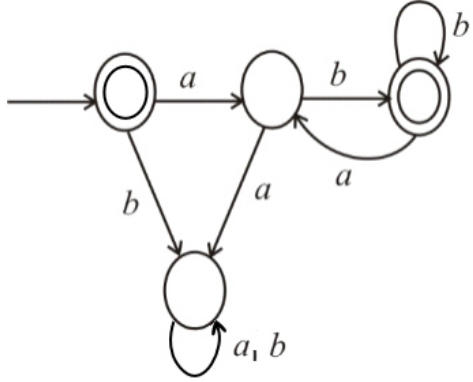
(e) The language is

$\overline{L} = \{ w \mid w$ is any string not in $(ab^+)^*\}$

$\overline{L}$ is the complement of a simpler language *L*.

Then the simple language is $L = \{ w \mid w$ is any string in $(ab^+)^*\}$

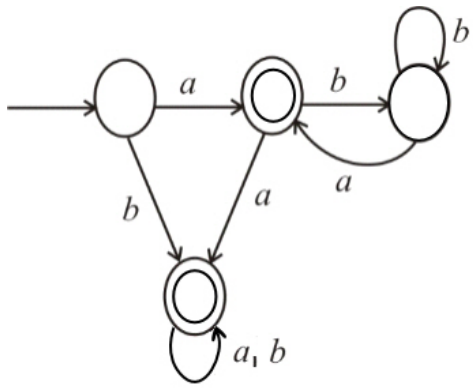
*DFA* that recognizes the language *L* is as follows:




Comments (7)

**Step 10** of 16

*DFA* that recognizes the language $\overline{L}$ is as follows:




(f)

Comments (5)

**Step 11** of 16

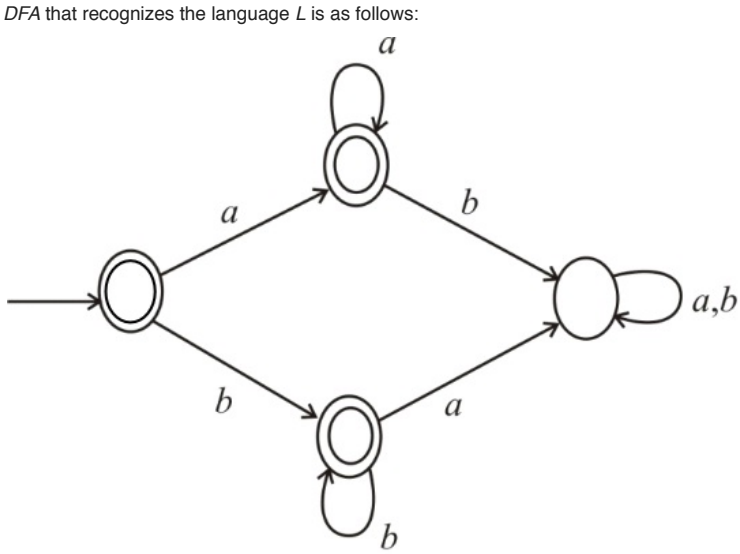
The language is

$\overline{L} = \{ w \mid w$ is any string not in $a^* \cup b^* \}$

$\overline{L}$ is the complement of a simpler language *L*.

Then the simple language is $L = \{ w \mid w$ is any string in $a^* \cup b^* \}$

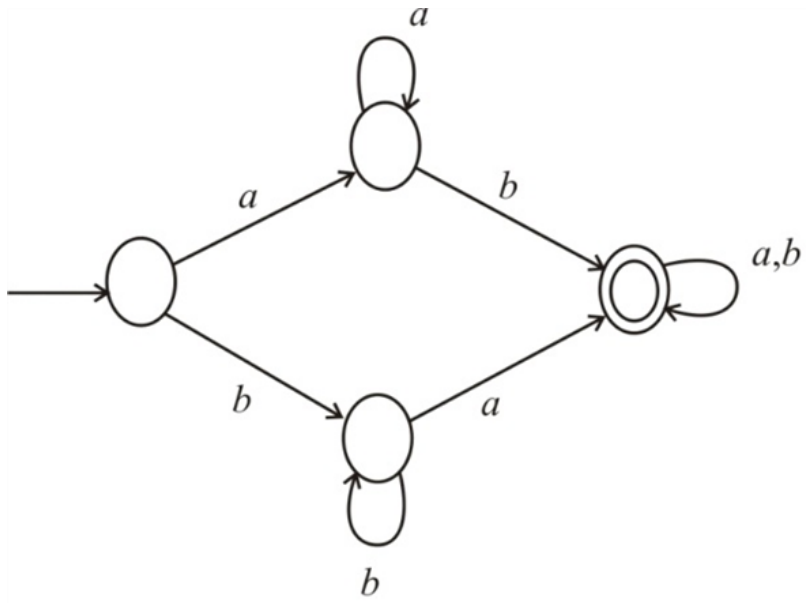
*DFA* that recognizes the language *L* is as follows:




Comments (3)

**Step 12** of 16

*DFA* that recognizes the language $\overline{L}$ is as follows:




Comment

**Step 13** of 16

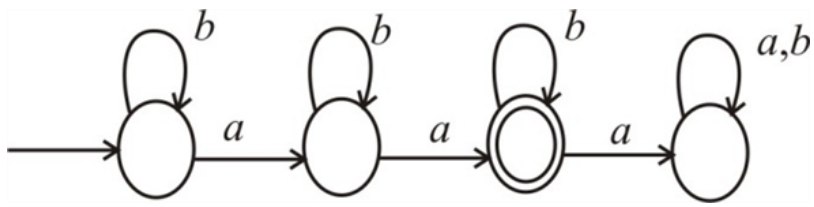
(g) The language is

$\overline{L}$ ={ $w$ | $w$ is any string that doesn't contain exactly two a's}

$\overline{L}$ is the complement of a simpler language *L*.

Then the simple language is $L$ ={ $w$ | $w$ is any string contain exactly two *a*'s}

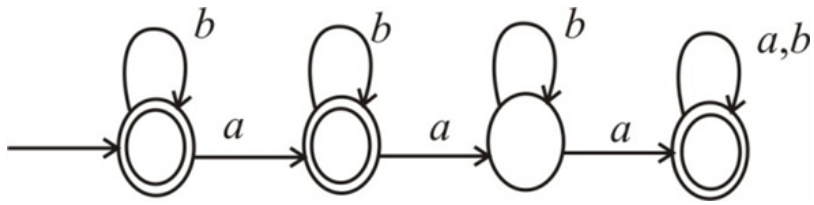
*DFA* that recognizes *L* is as follows:




Comment

**Step 14** of 16

*DFA* that recognizes $\overline{L}$ is as follows:




Comment

**Step 15** of 16

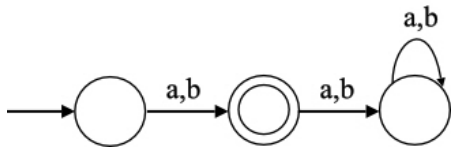
(h) The language is

$\overline{L}$ ={ $w$ | $w$ is any string except *a* and *b*}

$\overline{L}$ is the complement of a simpler language *L*.

Then the simple language is $L = \{ w \mid w$ is *a* and *b*$\}$

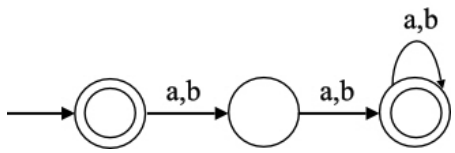
*DFA* that recognizes *L* is as follows:




Comments (6)

**Step 16** of 16

DFA that recognizes $\overline{L}$ is as follows:




Comment

## Problem

Give state diagrams of DFAs recognizing the following languages. In all parts, the alphabet is {0,1}.

**a.** {wl w begins with a 1 and ends with a 0}

**b.** {wl w contains at least three 1s}

**c.** {wl w contains the substring 0101 (i.e., w = x0101y for some x and y)}

**d.** {wl w has length at least 3 and its third symbol is a 0}

**e.** {wl w starts with 0 and has odd length, or starts with 1 and has even length}

**f.** {wl w doesn't contain the substring 110}

**g.** {wl the length of w is at most 5}

**h.** {wl w is any string except 11 and 111}

**i.** {wl every odd position of w is a 1}

**j.** {wl w contains at least two 0s and at most one 1}

**k.** { $\varepsilon$, 0}

**l.** {wl w contains an even number of 0s, or contains exactly two 1s}

**m.** The empty set

**n.** All strings except the empty string

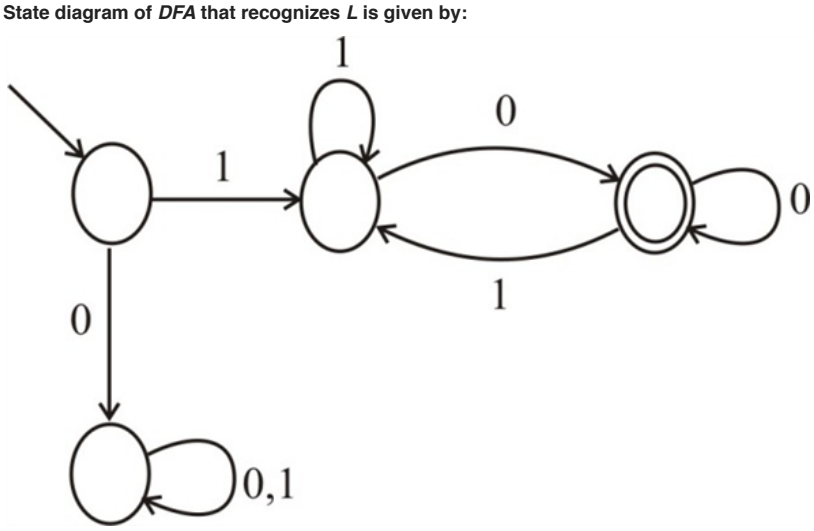
## Step-by-step solution

**Step 1** of 14

**(a)**

Language $L = \{ w \mid w$ begins with a 1 and ends with a 0}

**State diagram of *DFA* that recognizes *L* is given by:**



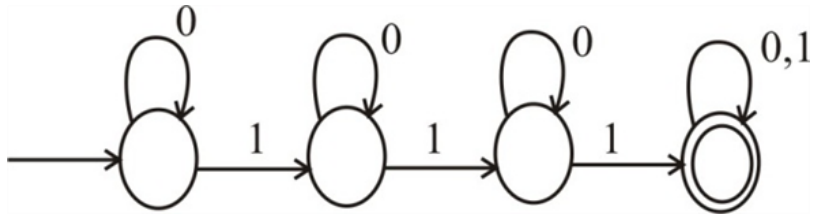

Comments (7)

**Step 2** of 14

**(b)**

Language $L = \{ w \mid w$ contains at least three 1s$\}$

**State diagram of *DFA* that recognizes *L* is given by:**



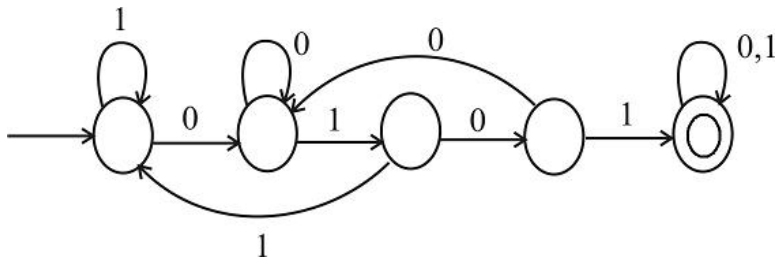

Comments (2)

**Step 3** of 14

**(c)**

Language $L = \{ w \mid w$ contains the substring 0101, i.e. $w = x0101y$ for some *x* and *y* $\}$

**State diagram of *DFA* that recognizes *L* is given by:**



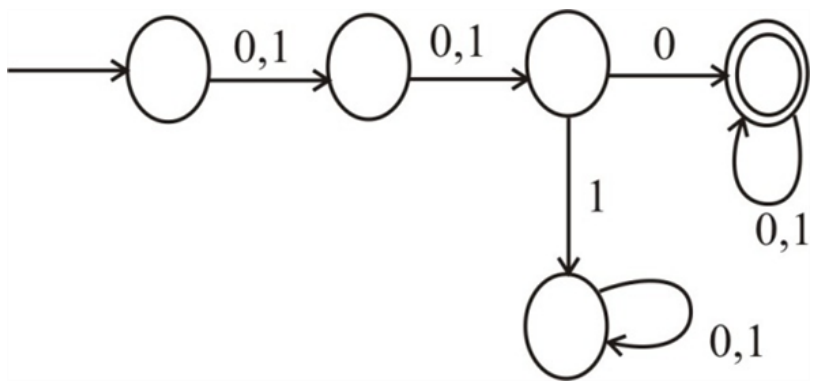

Comment

**Step 4** of 14

**(d)**

Language $L = \{ w \mid w$ has length at least 3 and its third symbol is a 0$\}$

**State diagram of *DFA* that recognizes *L* is given by:**



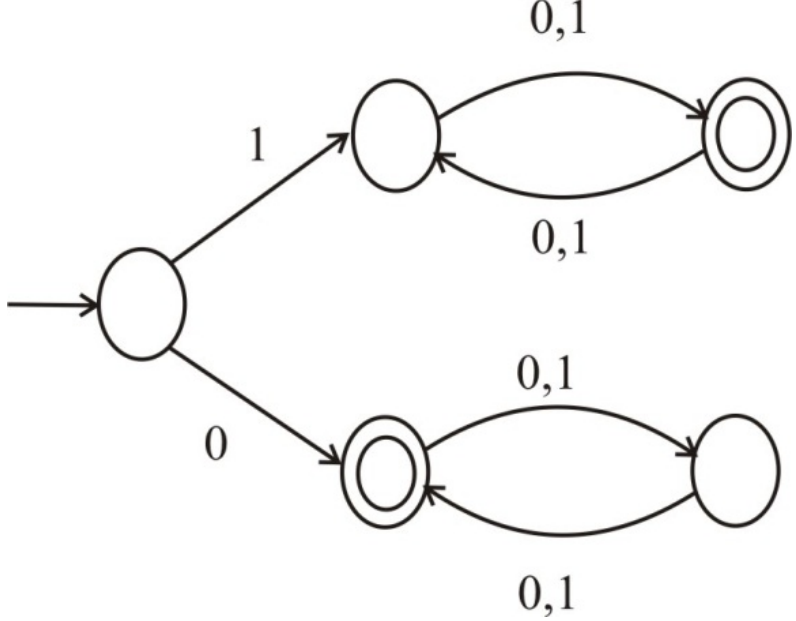

Comment

**Step 5** of 14

(e)

Language $L = \{ w \mid w$ starts with 0 and has odd length, or starts with 1 and has even length$\}$

**State diagram of *DFA* that recognizes *L* is given by:**



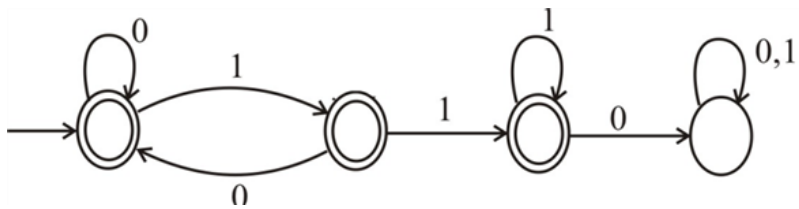

Comments (3)

**Step 6** of 14

**(f)**

Language $L = \{ w \mid w$ doesn't contain the substring 110$\}$

**State diagram of *DFA* that recognizes *L* is given by:**



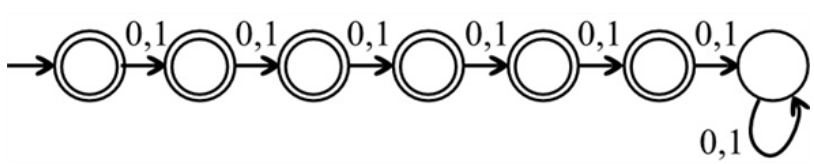

Comment

**Step 7** of 14

**(g)**

Language $L = \{w \mid$ length of $w$ is at most 5$\}$

**State diagram of *DFA* that recognizes *L* is given by:**



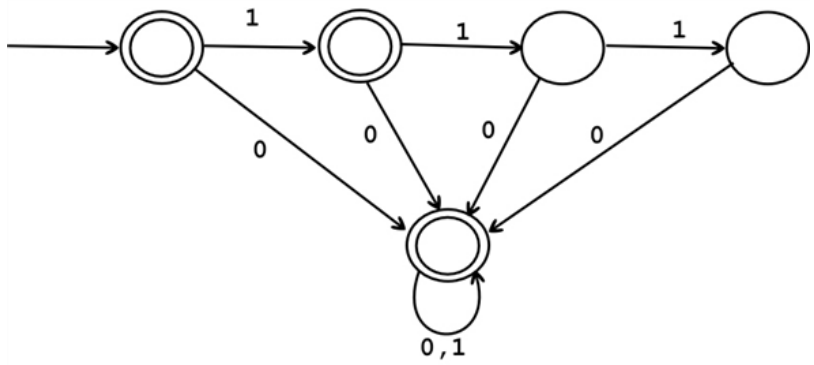

Comments (2)

**Step 8** of 14

**(h)**

Language $L = \{ w \mid w$ is any string except 11 and 111$\}$

**State diagram of *DFA* that recognizes *L* is given by:**



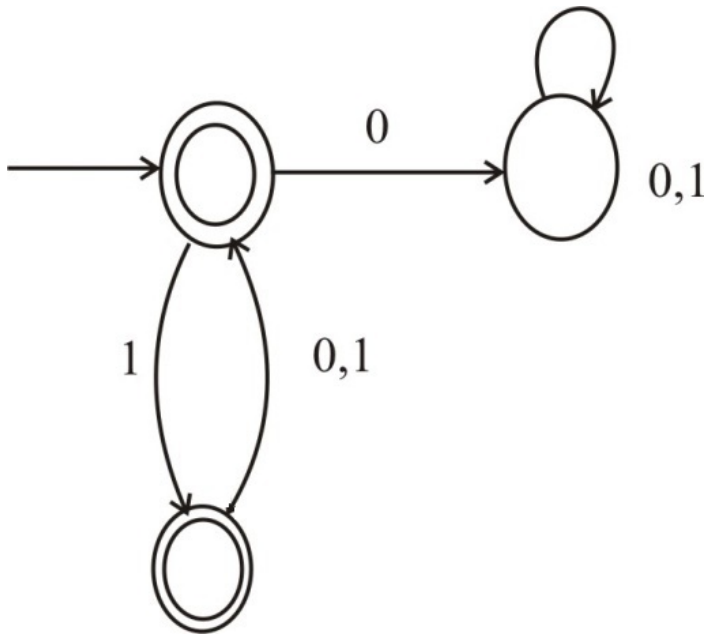

Comments (11)

**Step 9** of 14

**(i)**

Language $L = \{ w \mid w$ every odd position of $w$ is a 1$\}$

**State diagram of *DFA* that recognizes *L* is given by:**



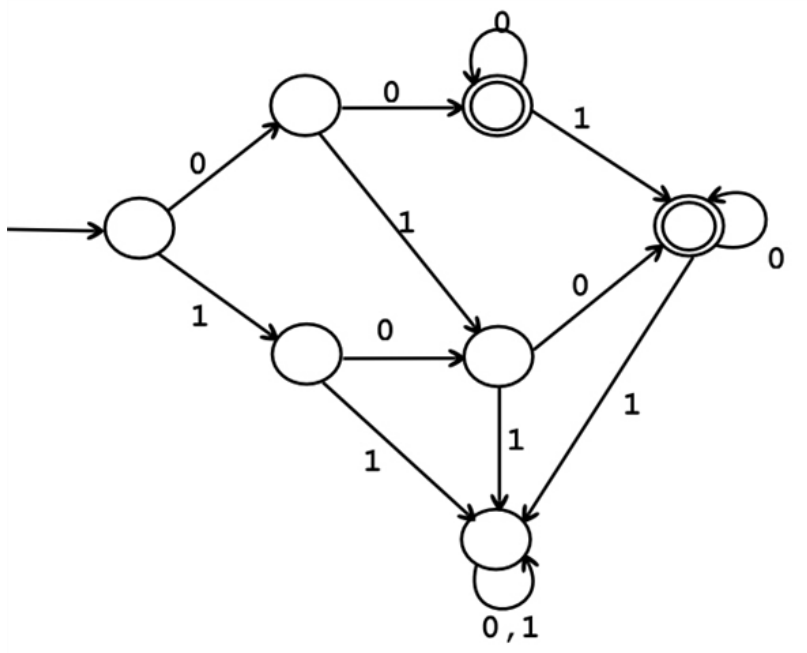

Comments (3)

**Step 10** of 14

(j)

Language $L = \{ w \mid w$contain at least two 0s and at most one 1$\}$

**State diagram of *DFA* that recognizes *L* is given by:**



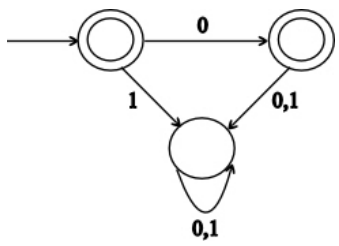

Comment

**Step 11** of 14

**(k)**

Language $L = \{\in, 0\}$

**State diagram of *DFA* that recognizes *L* is given by:**



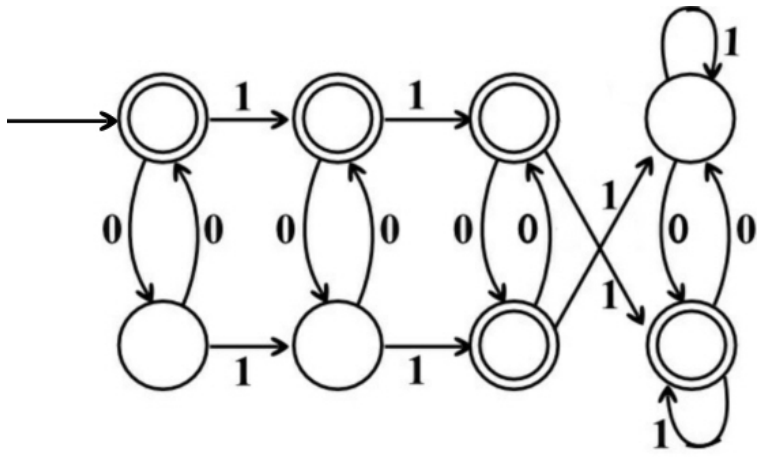

Comment

**Step 12** of 14

**(l)**

Language $L = \{\ w \mid w$ contains an even number of 0s, or contains exactly two 1s$\}$.

**State diagram of *DFA* that recognizes *L* is given by:**




The language L accepts the strings that contain even number of 0s or contains exactly two 1s. This language accepts the string if any one of the two conditions is satisfied.

Consider the string 11, the string 11 contains zero number of 0s which is even. In this case, the first condition is accepted. Thus, the language L accepts the string 11.

Comments (10)

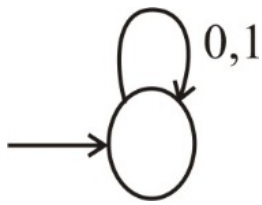
**Step 13** of 14

**(m)**

Language *L*= {The empty set}

The empty set does not contain the null string. The language *L* does not accept any string even the null string.

**State diagram of *DFA* that recognizes *L* is given by:**



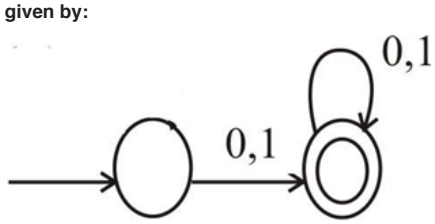

Comments (2)

**Step 14** of 14

(n)

Language $L =$ {all strings except the empty string}

**State diagram of *DFA* that recognizes *L* is given by:**




Comment

## Problem

Give state diagrams of NFAs with the specified number of states recognizing each of the following languages. In all parts, the alphabet is {0,1}.

A**a.** The language {wl w ends with 00} with three states

**b.** The language of Exercise 1.6c with five states

**c.** The language of Exercise 1.6l with six states

**d.** The language {0} with two states

**e.** The language $0^*1^*0^+$ with three states

A**f.** The language $1^*(001^+)^*$ with three states

**g.** The language { $\varepsilon$ } with one state

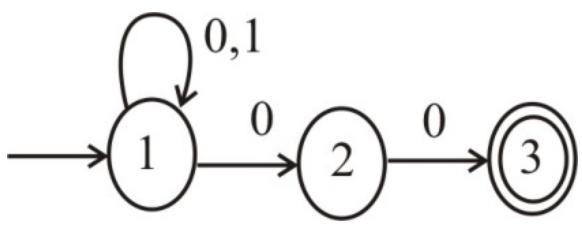
**h.** The language $0^*$ with one state

## Step-by-step solution

**Step 1** of 8

a.

Consider the Language $L = \{w \mid w \text{ ends with } 00\}$ with three states over the alphabet $\Sigma = \{0,1\}$. The language states that the Finite automata should consist of three states that accept the strings over the alphabet $\Sigma = \{0,1\}$ and ends with 00. Let *M* be the NFA that recognizes $L$. The state diagram of *M* is as follows:



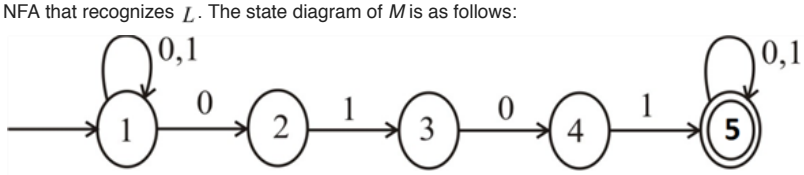

Comment

**Step 2** of 8

b.

Consider the Language

$L = \{w \mid w \text{ contains the substring } 0101 \text{ i.e, } w = x0101y \text{ for some } x \text{ and } y\}$ with five states over the alphabet $\Sigma = \{0,1\}$. The language states that the Finite automata should consist of five states that accept the strings over the alphabet $\Sigma = \{0,1\}$ and contains the substring 0101. Let *M* be the NFA that recognizes $L$. The state diagram of *M* is as follows:




Comment

**Step 3** of 8

c.

Consider the Language

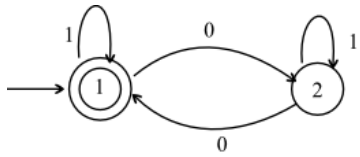
$L = \{w \mid w \text{ contains an even number of 0s, or contains exactly two 1s}\}$ with 6 states over the alphabet $\Sigma = \{0,1\}$. Let *M* be the NFA that recognizes $L$. Divide *L* into 2 languages $L_1$ and $L_2$.

$L_1 = \{w \mid w \text{ contains on even number of 0s}\}$

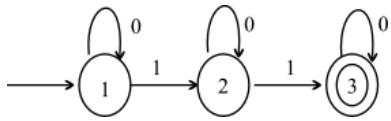
$L_2 = \{w \mid w \text{ contains exactly two 1s}\}$

Let $M_1, M_2$ be the NFAs recognizes $L_1, L_2$ respectively.

State diagram of $M_1$ is as follows:

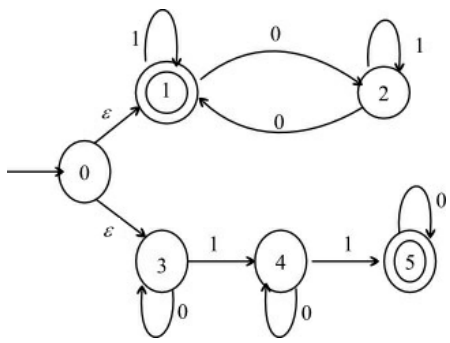



The state diagram of $M_2$ is as follows:




Now $L = L_1 \cup L_2$

The sate diagram of *M* that recognizes *L* is as follows:

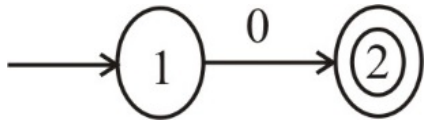



Comments (2)

**Step 4** of 8

d.

Consider the Language $L_1 = \{w \mid w \text{ contains only } 0\}$ with 2 states over the alphabet $\Sigma = \{0,1\}$. Let *M* be the NFA that recognizes $L_1$. The state diagram of *M* is as follows:

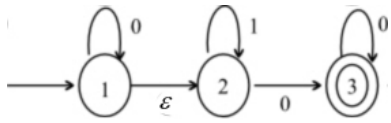



Comment

**Step 5** of 8

e.

Consider the Language $L = \{w \mid w \text{ contains only } 0^*1^*0^+\}$ with 3 states over the alphabet $\Sigma = \{0,1\}$. The language states that the finite automata accept all the strings containing any number of zeroes and ones followed by at least one zero. Let *M* be the NFA that recognizes $L$. The state diagram of *M* is as follows:

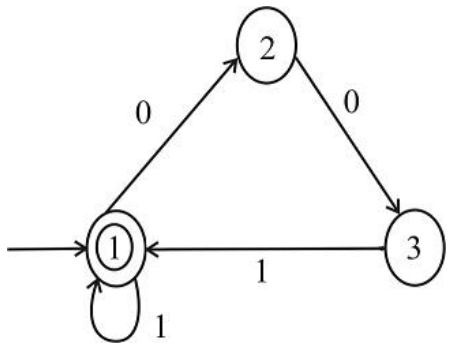



f.

Comments (21)

**Step 6** of 8

Consider the Language $L$ that accepts the strings of the form $1*(001^{+})*$ with 3 states over the alphabet $\Sigma = \{0,1\}$. Let *M* be the NFA that recognizes $L$. The state diagram of *M* is as follows:

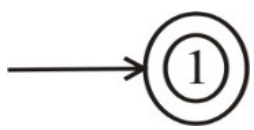



Comment

**Step 7** of 8

g.

Consider the Language $L_1 = \{w \mid w \text{ the language results in empty string } \varepsilon\}$ with one state over the alphabet $\Sigma = \{0,1\}$. The language states that the finite automata accept a null string. Let *M* be the NFA that recognizes $L_1$. The state diagram of *M* is as follows:

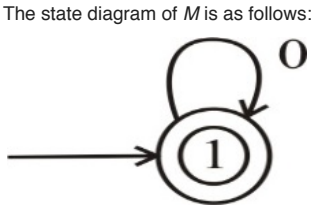



Comment

**Step 8** of 8

h.

Consider the Language *L* that accepts the strings of the form $0*$with one state over the alphabet $\Sigma = \{0,1\}$. Let *M* be the NFA that recognizes *L*. The state diagram of *M* is as follows:




Comment

## Problem

Use the construction in the proof of Theorem 1.45 to give the state diagrams of NFAs recognizing the union of the languages described in

**a.** Exercises 1.6a and 1.6b.

**b.** Exercises 1.6c and 1.6f.

THEOREM **1.45**

The class of regular languages is closed under the union operation.

## Step-by-step solution

**Step 1** of 9

a)

Consider the languages,

$L_1 = \{ w \mid w$ begins with 1 and ends with *a* 0 } and

$L_2 = \{ w \mid w$ contains at least three 1s} on $\Sigma = \{0,1\}$

$M_1$ be the NFA that recognizes $L_1$ and

$M_2$ be the NFA that recognizes $L_2$.

Comment

**Step 2** of 9

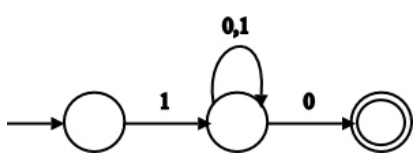
Let $L = L_1 \cup L_2$

Now *M* be the NFA that recognizes *L*.

- $L_1 = \{ w \mid w$ begins with *a* 1 and ends with *a* 0}

$L_1 = 1(0,1)^* 0$

The state diagram of $M_1$ that recognizes $L_1$ is



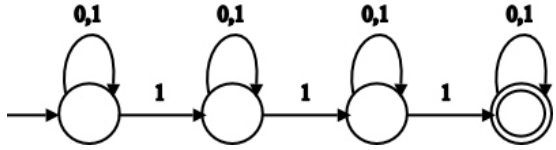

Comments (5)

**Step 3** of 9

- $L_2 = \{ w \mid w$ contains at least three 1s}

$L_2 = (0,1)^* 1 (0,1)^* 1 (0,1)^* 1 (0,1)^*$

The state diagram of $M_2$ that recognizes $L_2$ is

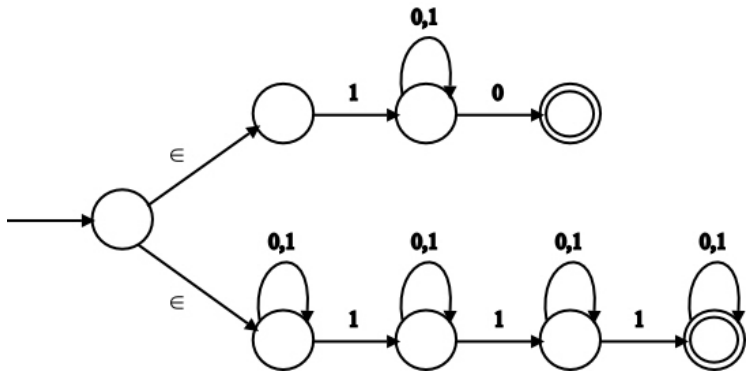



Comments (3)

**Step 4** of 9

Now *L* is union of $L_1$ and $L_2$.

So the state diagram of *M* that recognizes *L* is described as follows.




Comments (1)

**Step 5** of 9

(b) Languages are

$L_1 = \{w \mid w \text{ contains the substring } 0101 \text{ i.e, } w = x0101y \text{ for some } x \text{ and } y\}$ on $\sum = \{0,1\}$

$L_1 = \{w \mid w \text{ doesn't contain the substring } 110\}$ on $\Sigma = \{0,1\}$

$M_1$ be NFA that recognizes $L_1$ and

$M_2$ be the NFA that recognizes $L_2$

Comment

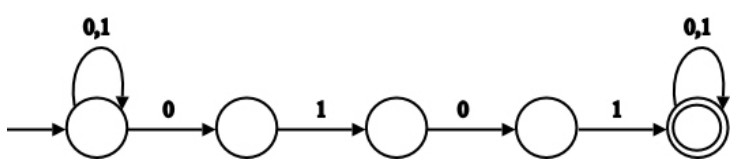
**Step 6** of 9

Let $L = L_1 \cup L_2$

Now *M* be the NFA that recognizes *L*.

- $L_1 = \{w \mid w \text{ contains the substring } 0101\}$

The state diagram of $M_1$ that recognizes $L_1$ is

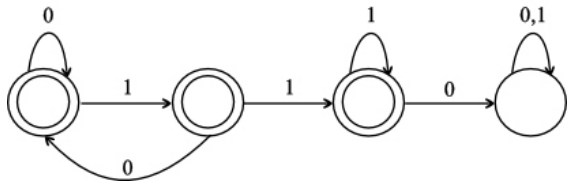



Comments (3)

**Step 7** of 9

- $L_2 = \{w \mid w \text{ doesn't contain the substring } 110\}$

First we give the state diagram of the machine which recognize the language machine which recognize the language




Comments (4)

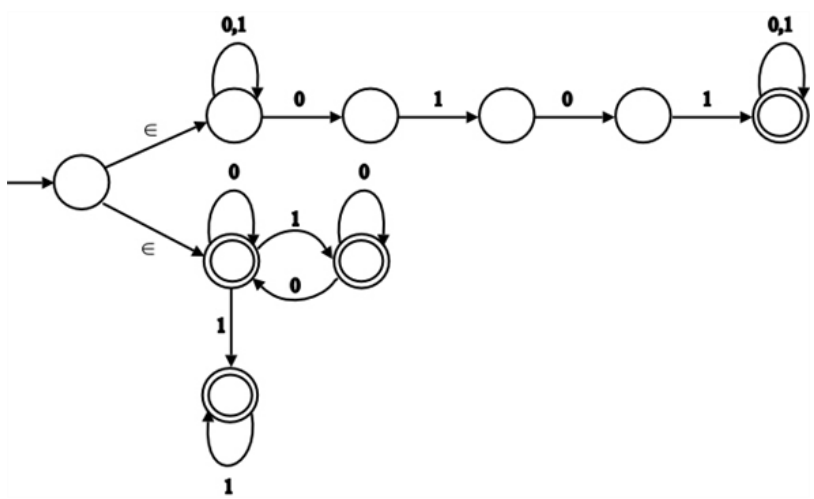
**Step 8** of 9

Now *L* is the union of $L_1$ and $L_2$

So the state diagram of *M* that recognizes *L* is described as follows

Comment

**Step 9** of 9




Comments (3)

## Problem

Use the construction in the proof of Theorem 1.47 to give the state diagrams of NFAs recognizing the concatenation of the languages described in

**a.** Exercises 1.6g and 1.6i.

**b.** Exercises 1.6b and 1.6m.

THEOREM **1.47**

The class of regular languages is closed under the concatenation operation.

## Step-by-step solution

**Step 1** of 8

(a) Languages are

$L_1 = \{w \mid \text{the length of } w \text{ is at most } 5\}$ on $\Sigma = \{0,1\}$

And $L_2 = \{w \mid \text{every odd position of } w \text{ is } a1\}$ on $\Sigma = \{0,1\}$

$M_1$ be the NFA that recognizes $L_1$ and

$M_2$ be the NFA that recognizes $L_2$.

Comment

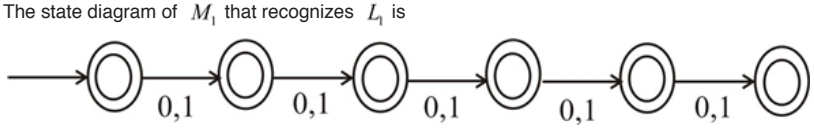
**Step 2** of 8

Let $L = L_1 0 L_2$

*M* be the NFA that recognizes *L*.

- $L_1 = \{w \mid \text{ the length of } w \text{ is at most } 5\}$

The state diagram of $M_1$ that recognizes $L_1$ is



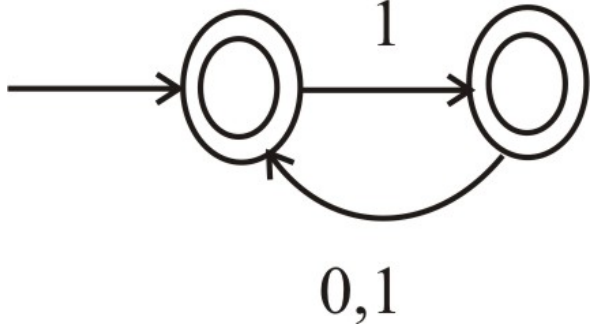

Comment

**Step 3** of 8

- $L_2 = \{w\} \text{ every odd position of } w \text{ is } a1\}$

$L_2 = (1\Sigma)^*$

The state diagram of $M_2$ that recognizes $L_2$ is

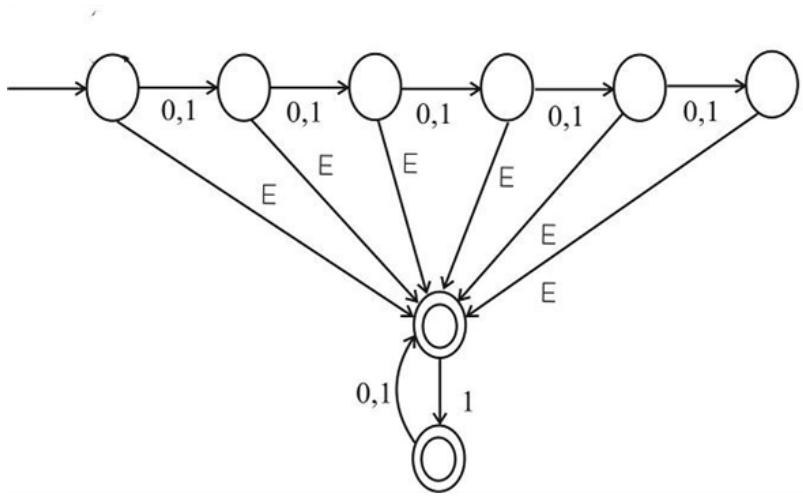



Comment

**Step 4** of 8

*L* is concatenation of $L_1$ and $L_2$

So the state diagram of $M$ that recognizes *L* is described as follows




Comment

**Step 5** of 8

(b) Given Languages are

$L_1 = \{w \mid w \text{ contains at least three 1s}\}$ on $\Sigma = \{0,1\}$

And $L_2 = \{w \mid w \text{ is a empty set}\}$ on $\Sigma = \{0,1\}$

$M_1$ be the NFA that recognizes $L_1$ and

$M_2$ be the NFA that recognizes $L_2$.

Comment

**Step 6** of 8

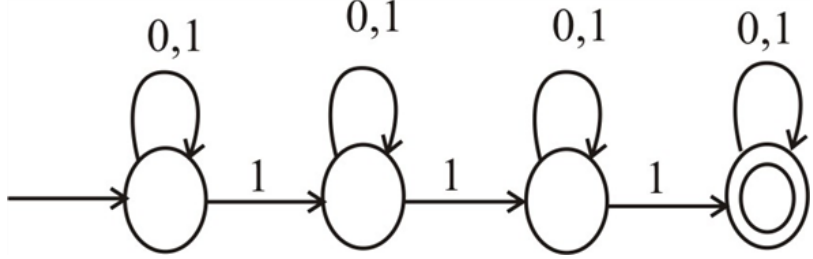
Let $L = L_1 0 L_2$

*M* be the NFA that recognizes *L*.

- $L_1 = \{w \mid w \text{ contains at least three 1s}\}$

$L_1 = (0,1)^* 1 (0,1)^* 1 (0,1)^* (0,1)^*$

The state diagram of $M_1$ that recognizes $L_1$ is



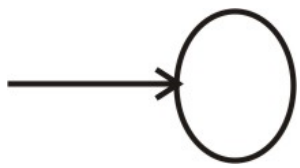

Comments (2)

**Step 7** of 8

- $L_2 = \{w \mid w \text{ is a empty set}\}$

$L_2 = \phi = \{\ \}$

The state diagram of $M_2$ that recognizes $L_2$ is

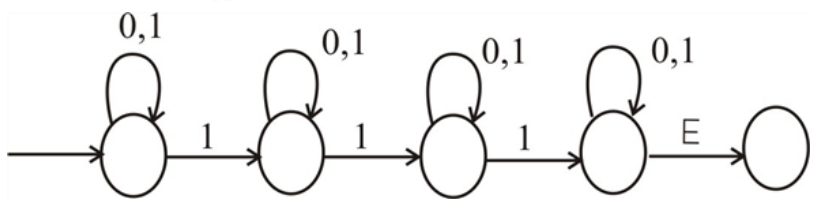


Comments (2)

**Step 8** of 8

*L* is concatenation of $L_1$ and $L_2$

So the state diagram of $M$ that recognizes *L* is described as follows




Comment

## Problem

Use the construction in the proof of Theorem 1.49 to give the state diagrams of NFAs recognizing the star of the languages described in

**a.** Exercise 1.6b.

**b.** Exercise 1.6j.

**c.** Exercise 1.6m.

THEOREM **1.49**

The class of regular languages is closed under the star operation.

## Step-by-step solution

**Step 1** of 3

(a) Language $L_1 = \{w \mid w \text{ contains at least three 1s}\}$

Let $M_1$ be the NFA that recognizes $L_1$.

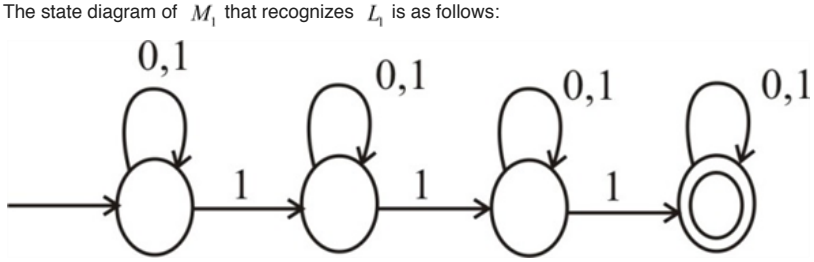
Let $L = L_1^*$

Let $M$ be the NFA that recognizes $L$.

$L_1 = \{w \mid w \text{ contains at least three 1s}\}$

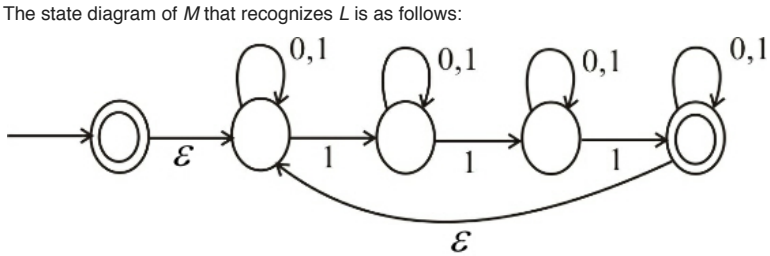
$L_1 = (0,1)^* 1 (0,1)^* 1 (0,1)^* 1 (0,1)^*$

The state diagram of $M_1$ that recognizes $L_1$ is as follows:




$L$ is the language that recognizes star of $L_1$

The state diagram of $M$ that recognizes $L$ is as follows:




Comments (7)

**Step 2** of 3

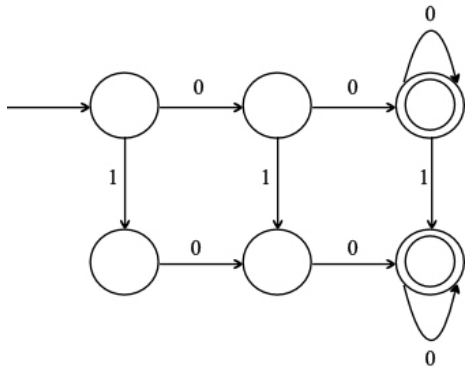
(b) Languages $L_1 = \{w \mid w \text{ contains at least two 0s and at most one 1}\}$

Let $M_1$ be the NFA that recognizes $L_1$.

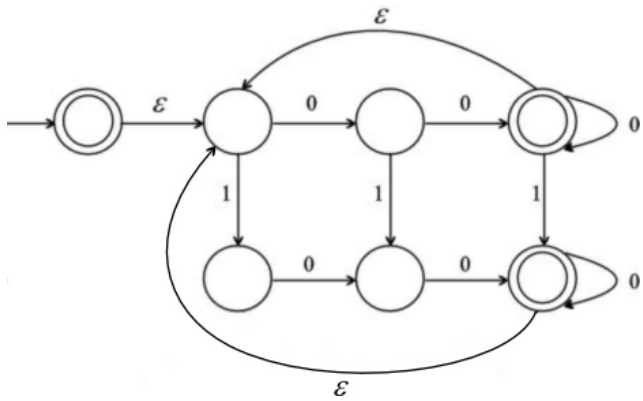
Let $L = L_1^*$

Let *M* be the NFA that recognizes *L*.

$L_1 = \{w \mid w \text{ contains atleast two 0s and at most one 1}\}$




The state diagram of *M* that recognizes *L* is as follows:




Comment

**Step 3** of 3

(c) Languages $L_1 =$ The empty set.

Let $M_1$ be the NFA that recognizes $L_1$.

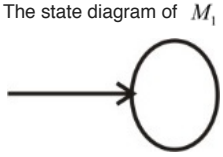
Let $L = L_1^*$

Let *M* be the NFA that recognizes *L*.

$L_1 =$ The empty set

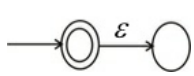
$L_1 = \phi = \{\ \}$

The state diagram of $M_1$ that recognizes $L_1$ is as follows:



$L$ is the star of $L_1$.

The state diagram of *M* that recognizes *L* is as follows:




Comment

## Problem

Prove that every NFA can be converted to an equivalent one that has a single accept state.

## Step-by-step solution

**Step 1** of 2

Let *M* be a NFA (non deterministic finite automatic)

Let *N* be the another NFA with single accept states $q_{\text{final}}$.

We go through every accept state *M* and do the following

(i) make it non – accepting state

(ii) add an $\in$-transition from that state to $q_{\text{final}}$

Then we will get NFA *N*.

If M has no accept states, then there will be no transitions coming into $q_{\text{final}}$.

Comment

**Step 2** of 2

Now we will discuss this formally.

$M = (Q, \Sigma, \delta, q_0, F)$ then

$N = (Q \cup \{q_{\text{final}}\}, \Sigma, \delta', q_0, \{q_{\text{final}}\})$ for any $q \in Q$ and $a \in \Sigma$

$$\delta'(q,a) = \begin{cases} \delta(q,a) \text{ if } a \neq \in \text{ or } q \notin F \\ \delta(q,a) \cup \{q_{\text{final}}\} \text{ if } a = \in \text{ and } q \in F \end{cases}$$

And $q'(q_{\text{final}}, a) = \phi$

Thus we get *N* by simply making every accept state of *M* as non – accepting state and adding an $\in -$ transition from that state to $q_{\text{final}}$

Thus M is equivalent to N.

Thus every NFA is converted to an equivalent one that has single accept state.

Comments (3)

## Problem

Let D = {wl w contains an even number of a's and an odd number of b's and does not contain the substring ab}. Give a DFA with five states that recognizes D and a regular expression that generates D. (Suggestion: Describe D more simply.)

## Step-by-step solution

**Step 1** of 4

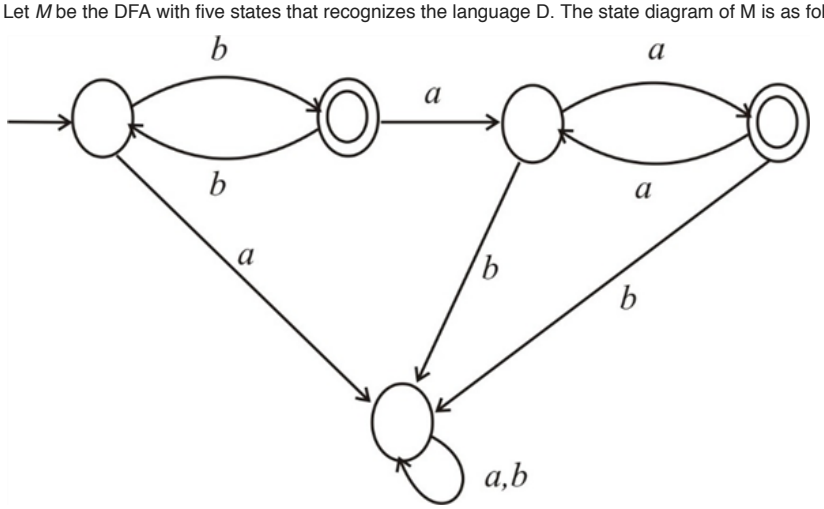
Consider the language $D = \{ w \mid w$ contains an even number of $a's$ and an odd number of $b's$ and does not contain the substring *ab*}.

The language D can be described simply as follows $D = \{ w \mid w$ contains an odd number of *b*'s followed by even number of *a*'s}.

Comment

**Step 2** of 4

Let *M* be the DFA with five states that recognizes the language D. The state diagram of M is as follows:




Comments (6)

**Step 3** of 4

The language accepts the strings like $\{b, baa, bbbaaaa, ...\}$. The string *b* is accepted by the language because, it contains the odd number of *b*'s (1) followed by even number of *a*'s (0).

Comment

**Step 4** of 4

Now, the language D can be expressed as combination of following two languages $D_1$ and $D_2$.

$D_1 = \{w \mid w \text{ contain odd number of b's}\}$

$D_2 = \{w \mid w \text{ contains even number of a's}\}$

$D = D_1 o D_2$

$R_1$ be the regular expression that generates $D_1$

$R_2$ be the regular expression that generates $D_2$

$R$ be the regular expression that generates $D$

$R = R_1 o R_2$

$R_1 = b(bb)^*$

$R_2 = (aa)^*$

$R = b(bb)^* o (aa)^*$

$R = b(bb)^*(aa)^*$

Therefore, the regular expression that generates the language $D$ is $b(bb)^*(aa)^*$.

Comment

## Problem

Let F be the language of all strings over {0,1} that do not contain a pair of 1s that are separated by an odd number of symbols. Give the state diagram of a DFA with five states that recognizes F. (You may find it helpful first to find a 4 state NFA for the complement of F.)

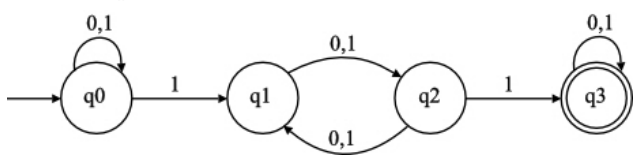
## Step-by-step solution

**Step 1** of 4

Consider the language *F* that accepts the strings over {0,1} that do not contain a pair of 1s that are separated by an odd number of symbols.

The complement of the language $\overline{F}$ accepts the strings over {0,1} that contain a pair of 1s that are separated by an odd number of symbols. The language accepts other strings as well.

The NFA for $\overline{F}$ is as follows:




The transition table for $\overline{F}$ is as follows:

| State | 0 | 1 |
|---|---|---|
| q0 | q0 | {q0,q1} |
| q1 | q2 | q2 |
| q2 | q1 | {q1,q3} |
| q3 | q3 | q3 |

Comment

**Step 2** of 4

Now, Convert the NFA to DFA.

The following are the steps to convert NFA to DFA:

**Step 1:** Create an empty set for the set of states of DFA. Initially $Q' = \phi$.

**Step 2:** Identify the start state in the NFA and make it as a start sate for the DFA. Add the initial state to the set of states of DFA.

**Step 3:** For every new state find the possible set of states for each input symbol using transition function of NFA and add new states to $Q'$. Start the procedure with the initial state.

**Step 4:** If no new states are found then stop the procedure. The final state of DFA will be all states which contain final states of NFA.

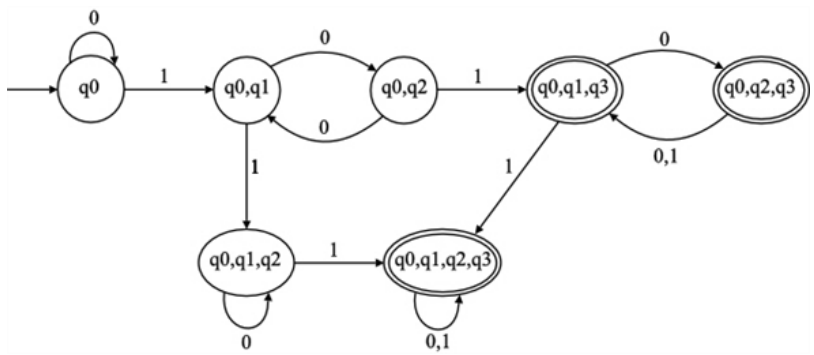
The transition table of the DFA is as follows:

| State | 0 | 1 |
|---|---|---|
| q0 | q0 | {q0,q1} |
| {q0,q1} | {q0,q2} | {q0,q1,q2} |
| {q0,q2} | {q0,q1} | {q0,q1,q3} |
| {q0,q1,q2} | {q0,q1,q2} | {q0,q1,q2,q3} |
| {q0,q1,q3} | {q0,q2,q3} | {q0,q1,q2,q3} |
| {q0,q1,q2,q3} | {q0,q1,q2,q3} | {q0,q1,q2,q3} |
| {q0,q2,q3} | {q0,q1,q3} | {q0,q1,q3} |

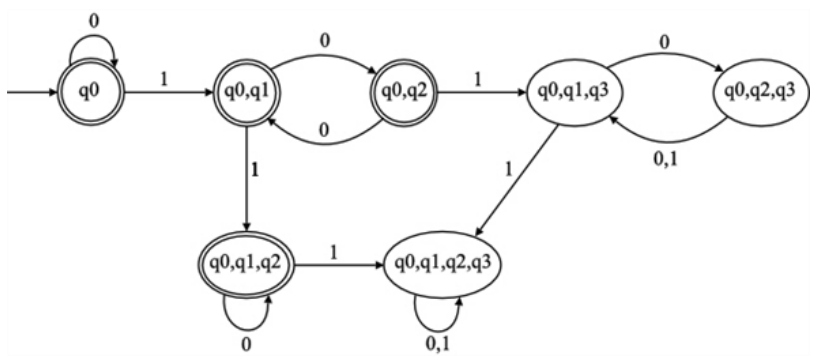
Comment

**Step 3** of 4

Draw the DFA using the above transition table.




The above DFA should be complemented to get the DFA for the language *F*. Change the final states to non-final states and vice versa.

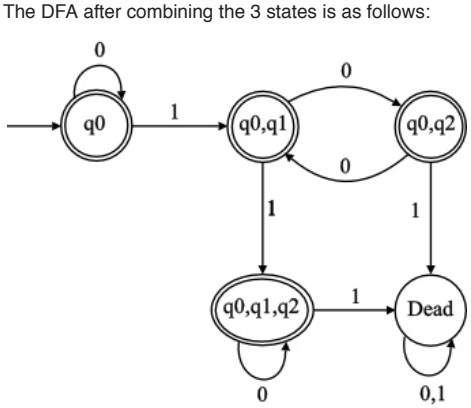



The states {q0,q1,q3}, {q0,q2,q3} and {q0,q1,q2,q3} are dead states. Combine all the three dead states into a single state.

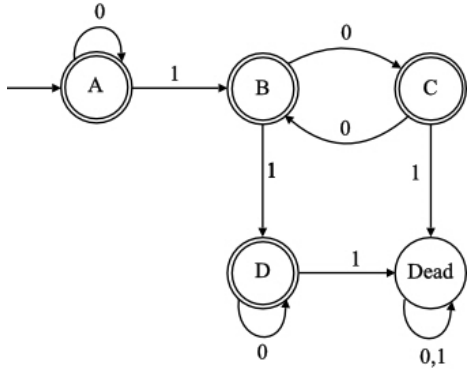
Comment

**Step 4** of 4

The DFA after combining the 3 states is as follows:




Rename the states in the above DFA for simplicity. The minimized DFA that accepts the language *F* is as follows:




Comments (2)

## Problem

**a.** Show that if M is a DFA that recognizes language B, swapping the accept and nonaccept states in M yields a new DFA recognizing the complement of B. Conclude that the class of regular languages is closed under complement.

**b.** Show by giving an example that if M is an NFA that recognizes language C, swapping the accept and nonaccept states in M doesn't necessarily yield a new NFA that recognizes the complement of C. Is the class of languages recognized by NFAs closed under complement? Explain your answer.

## Step-by-step solution

**Step 1** of 4

(a)

M is a DFA that recognizes the regular language *B*

Let $M'$ be the new DFA that has swapped accept and non accept states in M.

• Consider $M'$ accepts a string $x$.

• Run $M'$ on $x$ then $M'$ surely enters into accept state.

• The machines $M, M'$ have swapped accept and non-accept states.

• So if we run M on *x*, then M will end in a non-accept state.

• Thus if *x* is accepted by $M'$ then *x* does not accepted by *M*.

• Similarly if *x* is accepted by *M* then *x* does not accepted by $M'$.

• One can say that if $x \in B$ then $x \notin \overline{B}$ (complement of B) and vice – versa

• So $M'$ will accept the strings that are not accepted by *M*.

• Therefore $M'$ recognizes the languages which are complement of B.

• As M recognizes a regular language *B*, there exists $M'$ which recognizes complement of *B* which is also regular.

• Hence class of regular languages closed under complement.

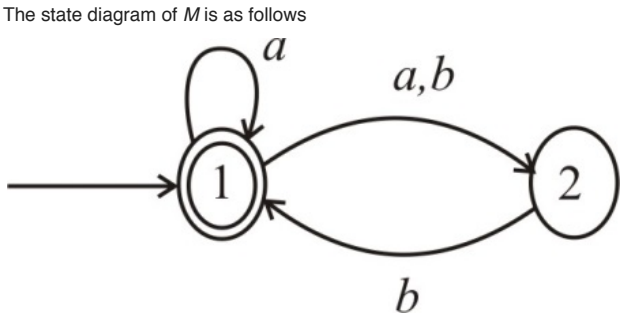
Comment

**Step 2** of 4

(b)

Let *M* be a NFA that recognizes a languages *C*.

The state diagram of *M* is as follows

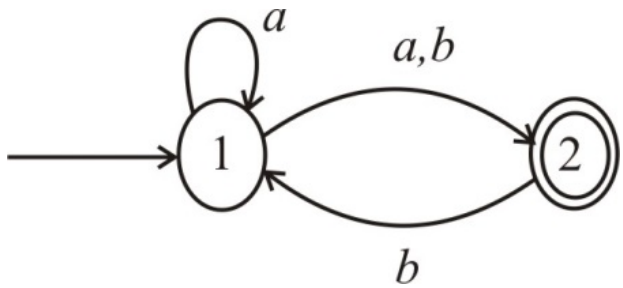



Comments (5)

**Step 3** of 4

Clearly the string a is accepted by *M*.

If we swap accept and non accept states of *M* then we will get the following $M'$




Comment

**Step 4** of 4

Clearly the string a is accepted by $M'$.

This shown that swapping accept and non - accept states of NFA doesn't necessarily yield a new NFA recognizing the complement of the original one.

Consider the facts,

"The class of languages recognized by NFAs is precisely the class of languages recognizes by DFAs" …… (a)

"The class of languages recognized by DFAs is closed under complement" …… (b)

Therefore from the facts (a) and (b)

Class of languages recognize by NFAs closed under complement.

Comment

## Problem

Give a counterexample to show that the following construction fails to prove Theorem 1.49, the closure of the class of regular languages under the star operation.7

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize $A_1$. Construct $N = (Q_1, \Sigma, \delta, q_1, F)$ as follows. $N$ is supposed to recognize $A_1^*$.

**a.** The states of $N$ are the states of $N_1$.

**b.** The start state of $N$ is the same as the start state of $N_1$.

**c.** $F = \{q_1\} \cup F_1$.
The accept states $F$ are the old accept states plus its start state.

**d.** Define $\delta$ so that for any $q \in Q_1$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \notin F_1 \text{ or } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon. \end{cases}$$

(Suggestion: Show this construction graphically, as in Figure 1.50.)

Figure 1.50

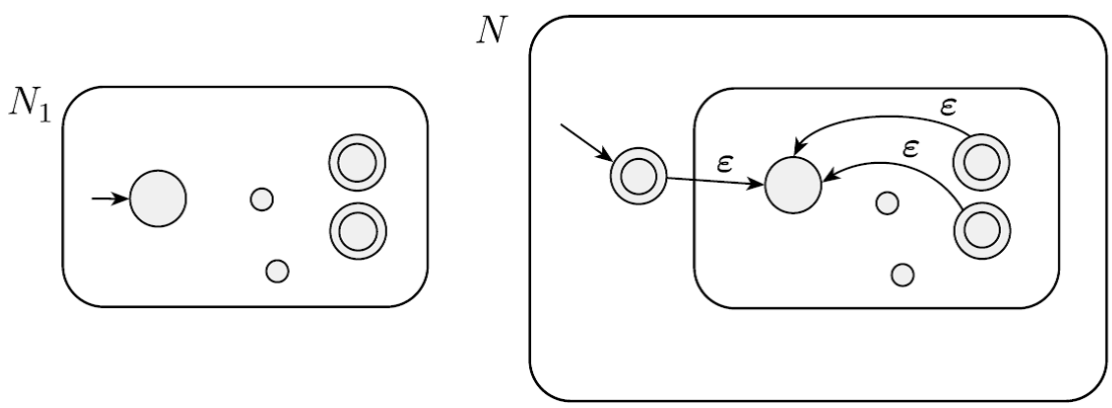



**FIGURE 1.50**
Construction of $N$ to recognize $A^*$

## Step-by-step solution

**Step 1** of 1

Theorem 1.49 states that "*The class of regular languages closed under the star operation*".

Consider the data,

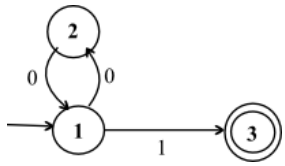
A language *A* is recognized by the automata $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$.

Assume *N* is the Non-deterministic finite automata which recognize the language $A_1^*$.

**Example:**

Assume a language $A_1 = \{(00)^* 1\}$.

The finite state automata $N_1$, which recognizes the language $A_1$ is as follows:




The following procedure is used to construct the finite state automata *N*, which recognizes the language $A_1^*$:

a. States of *N* are the states of $N_1$.

States of $N_1$ are {1,2,3}.

So, States of *N* are {1,2,3}.

b. The start state of *N* is same as start state of $N_1$.

Start state of $N_1$ is {1}.

So, start state of *N* is {1}.

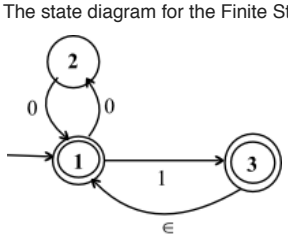
c. $F = \{q_1\} \cup F_1$. The accept states for F are the accept states of $F_1$ including the start state. So, the accept state for F are **1** and **3.**

d. Define the transition $\delta$ for any $q \in Q_1$ and any $a \in \Sigma_\epsilon$ by using the following transition:

$$\delta(q,a) = \begin{cases} \delta^1(q,a) & q \notin F_1 \text{ and } a \neq \epsilon \\ \delta^1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \epsilon \end{cases}$$

$$\begin{aligned} \delta(1,\epsilon) &= \delta_1(1,\epsilon) \cup \{1\} \\ &= \phi \cup \{1\} \\ &= \{1\} \\ \delta(3,\epsilon) &= \delta_1(3,\epsilon) \cup \{1\} \\ &= \phi \cup \{1\} \\ &= \{1\} \\ \delta(1,0) &= \delta_1(1,0) \\ &= 2 \\ \delta(1,1) &= \delta_1(1,1) \\ &= 3 \\ \delta(2,0) &= \delta_1(2,0) \\ &= 1 \\ \delta(2,1) &= \delta_1(2,1) \\ &= \phi \\ \delta(3,0) &= \delta_1(3,0) \\ &= \phi \\ \delta(3,1) &= \delta_1(3,1) \\ &= \phi \end{aligned}$$

The state diagram for the Finite State automata is as follows:




• The above finite automata adds the start state to the set of accept states, which adds some other undesired strings and $\epsilon$ to the recognized language .

• A new start state which is also an accept state is not added to the automata. Thus, the new state is not added to the automata and leads to different automata from original automata.

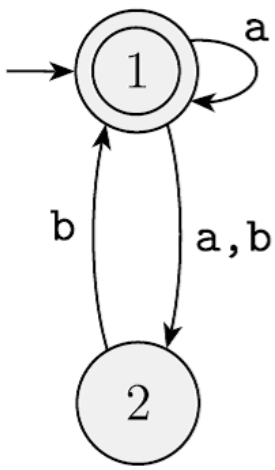
Comment

## Problem

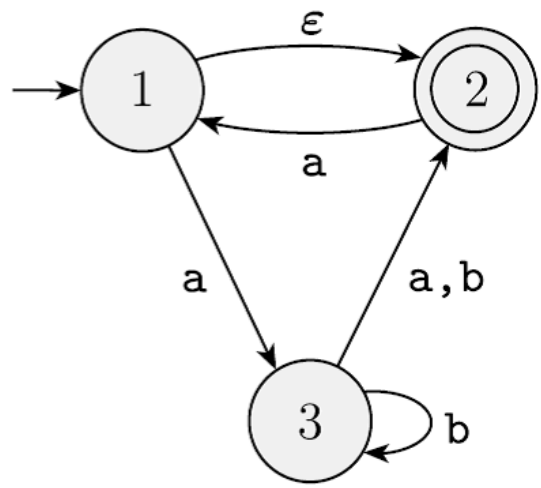
Use the construction given in Theorem 1.39 to convert the following two nondeterministic finite automata to equivalent deterministic finite automata.

THEOREM **1.39**

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.




(a)



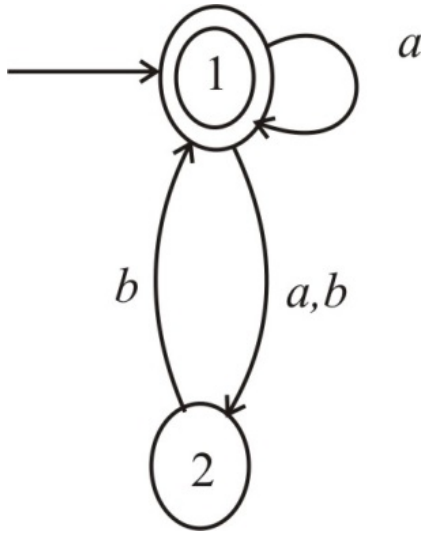

(b)

## Step-by-step solution

**Step 1** of 2

**a.**

Consider the Non-deterministic Finite Automata,



By using Theorem 1.39, "*For every non-deterministic finite automata, there is an equivalent Deterministic finite automation*".

**Constructing equivalent DFA for the given NFA:**

1. $Q^1 = P(Q)$ where $Q^1$ is the subset of all sets of *Q*.

So, $Q^1 = \{\phi, \{1\}, \{2\}, \{1,2\}\}$

2. For an element *R* in $Q^1$ and *a* in set of alphabets $\Sigma$, Calculate $\delta^1(R,a) = \{q \in Q \mid q \in \delta(r,a) \text{ for some } r \in R\}$. Here, $\delta^1$ performs the transition on *r* for some value of *a*.

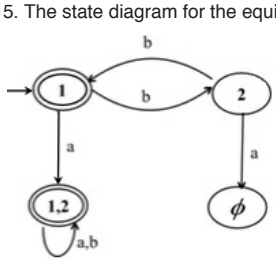
$$\begin{aligned}
\delta^1(\phi,a) &= \delta(\phi,a) \\
&= \phi \\
\delta^1(\phi,b) &= \delta^1(\phi,b) \\
&= \phi \\
\delta^1(\{1\},a) &= \delta(1,a) \\
&= \{1,2\} \\
\delta^1(\{1\},b) &= \delta(1,b) \\
&= \{2\} \\
\delta^1(\{2\},a) &= \delta(2,a) \\
&= \phi \\
\delta^1(\{2\},b) &= \delta(2,b) \\
&= \{1\} \\
\delta^1(\{1,2\},a) &= \delta(\{1,2\},a) \\
&= \delta(1,a) \cup \delta(2,a) \\
&= \{1,2\} \cup \phi \\
&= \{1,2\} \\
\delta^1(\{1,2\},b) &= \delta(\{1,2\},b) \\
&= \delta(1,b) \cup \delta(2,b) \\
&= \{2\} \cup \{1\} \\
&= \{1,2\}
\end{aligned}$$

3. $q_0' = \{q_0\}$ where $q_0$ is the start state in NFA.

Here, $q_0'$={1}.

4. $F' = \{R \in Q' \mid R \text{ contains an accept state of NFA}\}$. The machine M accepts the possible states where the NFA is present in the accept state.

5. The state diagram for the equivalent DFA is as follows:

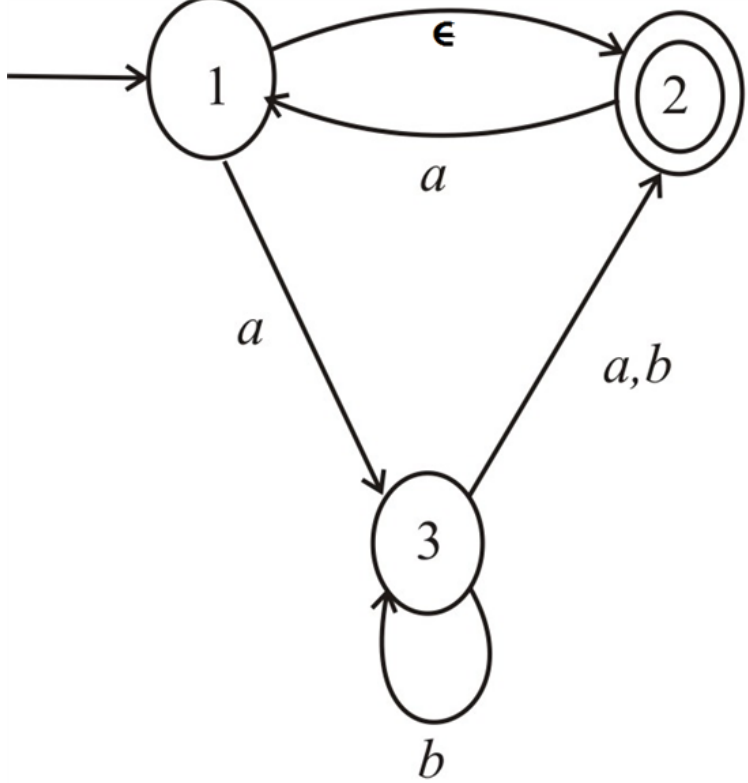



Comments (2)

**Step 2** of 2

**b.**

Consider the Non-deterministic Finite Automata,




By using Theorem 1.39, "*For every non-deterministic finite automata, there is an equivalent Deterministic finite automation*".

**Constructing equivalent DFA for the given NFA:**

The initial state of DFA is 1 let x= ($Q_x$, $\Sigma$, $\delta_x$, $q_0$, $F_x$ ).

1. $Q^1 = P(Q)$ where $Q^1$ is the subset of all sets of *Q*.

So, $Q^1 = \{\phi, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

2. Considering $\in$ notations for each $R \subseteq Q$.

$E(R) = \{q \mid q \text{ can be reached from } R \text{ by traveling along or more } \in \text{ arrows}\}$

The collection of states reached from R by moving along the $\in$notations is,

$$
\begin{aligned}
E(\phi) &= \phi \\
E(\{1\}) &= \{1,2\} \\
E(\{2\}) &= \{2\} \\
E(\{3\}) &= \{3\} \\
E(\{1,2\}) &= \{1,2\} \\
E(\{1,3\}) &= \{1,2,3\} \\
E(\{2,3\}) &= \{2,3\} \\
E(\{1,2,3\}) &= \{1,2,3\}
\end{aligned}
$$

3. Calculate $\delta^1(R,a) = \{q \in Q \mid q \in \delta(r,a) \ \ for\, some\, r \in R\}$. Here, $\delta^1$ performs the transition on *r* for some value of *a*.

$$
\begin{aligned}
\delta'(\phi,a) &= \phi \\
\delta'(\phi,b) &= \phi \\
\delta'(\{1\},a) &= E(\delta(1,a)) \\
&= E(\{3\}) \\
&= \{3\} \\
\delta'(\{1\},b) &= E(\delta(1,b)) \\
&= E(\phi) \\
&= \phi \\
\delta'(\{2\},a) &= E(\delta(1,a)) \\
&= E(\{1\}) \\
&= \{1,2\} \\
\delta'(\{2\},b) &= E(\delta(2,b)) \\
&= E(\phi) \\
&= \phi
\end{aligned}
$$

$$\begin{aligned}\delta'(\{3\},a) &= E(\delta(3,a)) \\ &= E(\{2\}) \\ &= \{2\}\end{aligned}$$

$$\begin{aligned}\delta'(\{3\},b) &= E(\delta(3,b)) \\ &= E(\{2,3\}) \\ &= \{2,3\}\end{aligned}$$

$$\begin{aligned}\delta'(\{1,2\},a) &= E(\delta(1,a)) \cup E(\delta(2,a)) \\ &= E(\{3\}) \cup E(\{1\}) \\ &= \{3\} \cup \{1,2\} \\ &= \{1,2,3\}\end{aligned}$$

$$\begin{aligned}\delta'(\{1,2\},b) &= E(\delta(1,b)) \cup E(\delta(2,b)) \\ &= E(\{\phi\}) \cup E(\{\phi\}) \\ &= \phi \cup \phi \\ &= \phi\end{aligned}$$

$$\begin{aligned}\delta'(\{1,3\},a) &= E(\delta(1,a)) \cup E(\delta(3,a)) \\ &= E(\{3\}) \cup E(\{2\}) \\ &= \{3\} \cup \{2\} \\ &= \{3\} \cup \{2\} \\ &= \{2,3\}\end{aligned}$$

$$\begin{aligned}\delta'(\{1,3\},b) &= E(\delta(1,b)) \cup E(\delta(3,b)) \\ &= E(\{\phi\}) \cup E(\{2,3\}) \\ &= \phi \cup \{2,3\} \\ &= \{2,3\}\end{aligned}$$

$$\begin{aligned}\delta'(\{2,3\},a) &= E(\delta(2,a)) \cup E(\delta(3,a)) \\ &= E(\{1\}) \cup E(\{2\}) \\ &= \{1,2\} \cup \{2\} \\ &= \{1,2\}\end{aligned}$$

$$\begin{aligned}\delta'(\{2,3\},b) &= E(\delta(2,b)) \cup E(\delta(3,b)) \\ &= E(\phi) \cup E(\{2,3\}) \\ &= \phi \cup \{2,3\} \\ &= \{2,3\}\end{aligned}$$

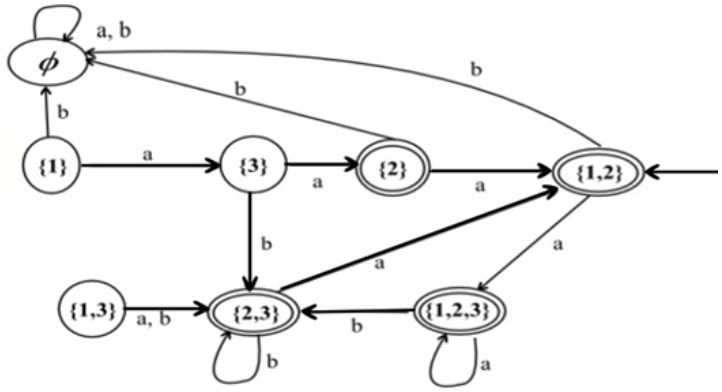
$$\begin{aligned}\delta'(\{1,2,3\},a) &= E(\delta(1,a)) \cup E(\delta(2,a)) \cup E(\delta(3,a)) \\ &= E(\{3\}) \cup E(\{1\}) \cup E(\{2\}) \\ &= \{3\} \cup \{1,2\} \cup \{2\} \\ &= \{1,2,3\}\end{aligned}$$

$$\begin{aligned}\delta'(\{1,2,3\},b) &= E(\delta(1,b)) \cup E(\delta(2,b)) \cup E(\delta(3,b)) \\ &= E(\phi) \cup E(\phi) \cup E(\{2,3\}) \\ &= \phi \cup \phi \cup \{2,3\} \\ &= \{2,3\}\end{aligned}$$

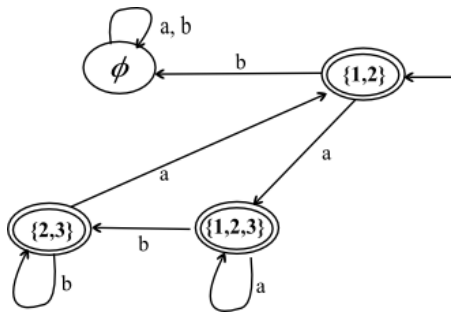
4. Changing $q_0'$ to $E(q_0)$ the start state becomes,

$$\begin{aligned}q_0^1 &= E(q_0) \\ q_0^1 &= E(\{1\}) \\ q_0^1 &= \{1,2\}\end{aligned}$$

5. The state diagram for the equivalent DFA is as follows:




Simplifying the machine by eliminating no arrow points. Here {1},{2},{1,3} and {3} do not contain any incoming arrows. Thus, the simplified machine is:




Comments (7)

## Problem

**a.** Give an NFA recognizing the language $(01 \cup 001 \cup 010)^*$.

**b.** Convert this NFA to an equivalent DFA. Give only the portion of the DFA that is reachable from the start state.

## Step-by-step solution

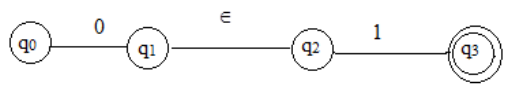
**Step 1** of 2

(a) Given Language $L = (01 \cup 001 \cup 010)^*$

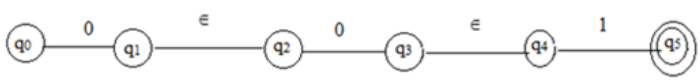
Assume that *M* as the NFA that recognizes language *L*.

The NFA *M* for the given language $L = (01 \cup 001 \cup 010)^*$ is as follows:

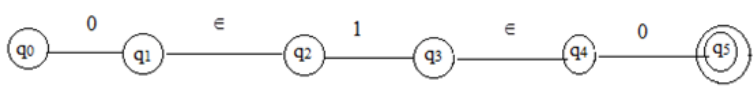
For string 01



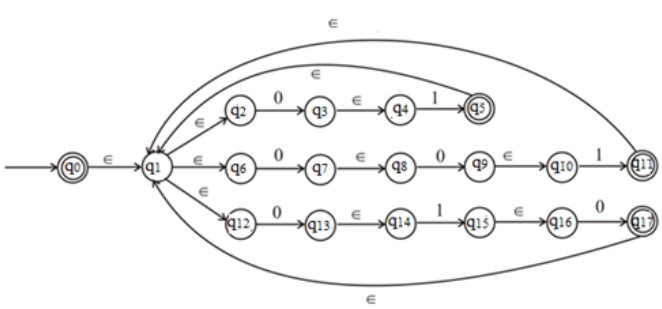

For string 001




For string 010




By joining all the above strings the final NFA for the $L = (01 \cup 001 \cup 010)^*$ is shown below:



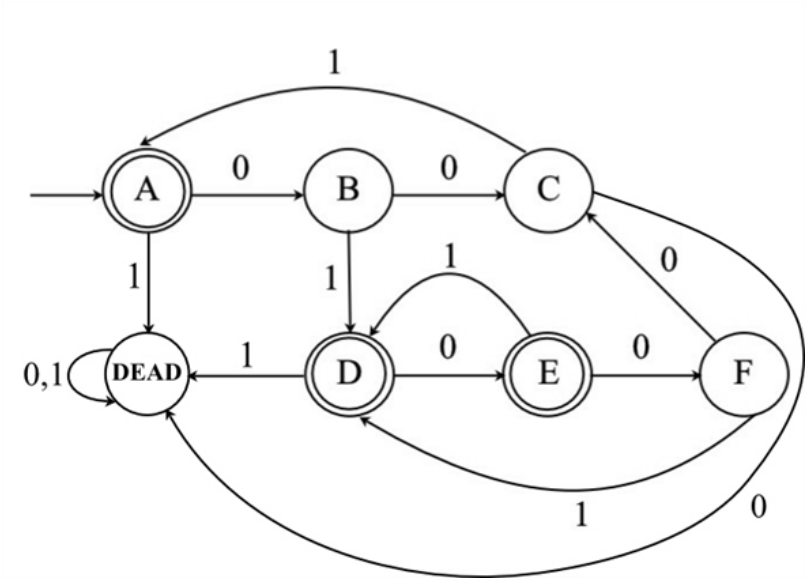

Comments (3)

**Step 2** of 2

(b) Conversion of the NFA to DFA.

First remove all the $\in$ symbols in the NFA *M* and draw the transitions that are present in *M.*

The equivalent DFA for NFA for the Language $L = (01 \cup 001 \cup 010)^*$ is as follows:




Comments (2)

## Problem

Give regular expressions generating the languages of Exercise 1.6.

Exercise 1.6.

Give state diagrams of DFAs recognizing the following languages. In all parts, the alphabet is {0,1}.

**a.** {wl w begins with a 1 and ends with a 0}

**b.** {wl w contains at least three 1s}

**c.** {wl w contains the substring 0101 (i.e., w = x0101y for some x and y)}

**d.** {wl w has length at least 3 and its third symbol is a 0}

**e.** {wl w starts with 0 and has odd length, or starts with 1 and has even length}

**f.** {wl w doesn't contain the substring 110}

**g.** {wl the length of w is at most 5}

**h.** {wl w is any string except 11 and 111}

**i.** {wl every odd position of w is a 1}

**j.** {wl w contains at least two 0s and at most one 1}

**k.** { $\varepsilon$ , 0}

**l.** {wl w contains an even number of 0s, or contains exactly two 1s}

**m.** The empty set

**n.** All strings except the empty string

## Step-by-step solution

**Step 1** of 14

In the regular expressions, '*' indicates that the preceding regular expression may appear zero or more times and '+' indicates that the preceding regular expression may appear one or more times.

a.

Consider the language $L = \{w \mid w \text{ begins with a 1 and ends with a 0}\}$ over the alphabet $\Sigma = \{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= 1\Sigma^*0 \\ &= 1(0+1)^*0 \end{aligned}$$

The strings accepted by the regular expression are $10, 100, 110, 1010, 1100, 10100, \ldots$

**Therefore, the regular expression is** $1(0+1)^*0$ **.**

Comments (1)

**Step 2** of 14

b.

Consider the language $L = \{w \mid w \text{ contains at least three 1s}\}$ over the alphabet $\Sigma = \{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= \Sigma^*1\,\Sigma^*1\Sigma^*1\Sigma^* \\ &= (0+1)^*1(0+1)^*1(0+1)^*1(0+1)^* \end{aligned}$$

The strings accepted by the regular expression are $111, 010101, 01101, 00001111, \ldots$

**Therefore, the regular expression is** $(0+1)^*1(0+1)^*1(0+1)^*1(0+1)^*$ **.**

Comments (4)

**Step 3** of 14

c.

Consider the language $L=\{w \mid w \text{ contains the substring } 0101\}$ over the alphabet $\Sigma=\{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= \Sigma^*0101\Sigma^* \\ &= (0+1)^*0101(0+1)^* \end{aligned}$$

The strings accepted by the regular expression are $0101, 001011, 101011, 1101010, \ldots$

**Therefore, the regular expression is** $(0+1)^*0101(0+1)^*$ **.**

Comments (1)

**Step 4** of 14

d.

Consider the language $L=\{w \mid w \text{ has length at least 3 and its third symbol is a } 0\}$ over the alphabet $\Sigma=\{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= \Sigma\Sigma 0\Sigma^* \\ &= (0+1)(0+1)0(0+1)^* \end{aligned}$$

The strings accepted by the regular expression are $000, 1101, \ldots$

**Therefore, the regular expression is** $(0+1)(0+1)0(0+1)^*$ **.**

Comment

**Step 5** of 14

e.

Consider the language,

$L=\{w \mid w \text{ starts with 0 and has odd length, or start with 1 and has even length}\}$ over the alphabet $\Sigma=\{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= 0(\Sigma\Sigma)^*+1\Sigma(\Sigma\Sigma)^* \\ &= 0((0+1)(0+1))^*+1(0+1)((0+1)(0+1))^* \end{aligned}$$

The strings accepted by the regular expression are $0, 011, 010, 00101, 10, 11, 1001, \ldots$

**Therefore, the regular expression is** $0((0+1)(0+1))^*+1(0+1)((0+1)(0+1))^*$ **.**

Comment

**Step 6** of 14

f.

Consider the language $L=\{w \mid w \text{ doesn't contain the substring } 110\}$ over the alphabet $\Sigma=\{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$R = 0^*\left(10^+\right)^*1^*$$

The strings accepted by the regular expression are $010, 011, 0101, \ldots$

**Therefore, the regular expression is** $0^*\left(10^+\right)^*1^*$ **.**

Comments (4)

**Step 7** of 14

g.

Consider the language $L = \{w \mid \text{the length of } w \text{ is at most } 5\}$ over the alphabet $\Sigma = \{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= \varepsilon + \Sigma + \Sigma\Sigma + \Sigma\Sigma\Sigma + \Sigma\Sigma\Sigma\Sigma + \Sigma\Sigma\Sigma\Sigma\Sigma \\ &= \varepsilon + (0+1) + (0+1)(0+1) + (0+1)(0+1)(0+1) + (0+1)(0+1)(0+1)(0+1) + (0+1)(0+1)(0+1)(0+1)(0+1) \\ &= \varepsilon + (0+1) + (0+1)^2 + (0+1)^3 + (0+1)^4 + (0+1)^5 \end{aligned}$$

The strings accepted by the regular expression are $\varepsilon, 0, 01, 101, 1010, 00000, \ldots$ The empty string is of length 0. The language accepts the strings of length from 0 to 5.

**Therefore, the regular expression is** $\varepsilon + (0+1) + (0+1)^2 + (0+1)^3 + (0+1)^4 + (0+1)^5$ .

Comments (1)

**Step 8** of 14

h.

Consider the language $L = \{w \mid w \text{ is any string except } 11 \text{ and } 111\}$ over the alphabet $\Sigma = \{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= \varepsilon + \Sigma + 0\Sigma + 10 + 0\Sigma\Sigma + 10\Sigma + 110 + \Sigma^3\Sigma^+ \\ &= \varepsilon + (0+1) + 0(0+1) + 10 + 0(0+1)(0+1) + 10(0+1) + 110 + (0+1)^3(0+1)^+ \end{aligned}$$

The strings accepted by the regular expression are $\varepsilon, 101, 110, 1010, \ldots$

**Therefore, the regular expression is,**

$$\varepsilon + (0+1) + 0(0+1) + 10 + 0(0+1)(0+1) + 10(0+1) + 110 + (0+1)^3(0+1)^+ .$$

Comments (4)

**Step 9** of 14

i.

Consider the language $L = \{w \mid \text{every odd position of } w \text{ is a } 1\}$ over the alphabet $\Sigma = \{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= (1\Sigma)^*(\varepsilon + 1) \\ &= (1(0+1))^*(\varepsilon + 1) \end{aligned}$$

The strings accepted by the regular expression are $\varepsilon, 101, 111, 1010, \ldots$

**Therefore, the regular expression is** $(1(0+1))^*(\varepsilon + 1)$ .

Comments (3)

**Step 10** of 14

j.

Consider the language $L = \{w \mid w \text{ contains at least two 0s and at most one } 1\}$ over the alphabet $\Sigma = \{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$R = 00^*00^*(\varepsilon + 1) + 00^*(\varepsilon + 1)00^* + (\varepsilon + 1)00^*00^*$$

The strings accepted by the regular expression are $001, 010, 100, \ldots$ In the first part of the regular expression $00^*00^*(\varepsilon + 1)$, there are two mandatory zeros and at most one 1. The optional 1 may appear at the start or middle or at the end. There are three parts in the regular expression to accept such strings.

**Therefore, the regular expression is** $00^*00^*(\varepsilon + 1) + 00^*(\varepsilon + 1)00^* + (\varepsilon + 1)00^*00^*$ .

Comments (1)

**Step 11** of 14

k.

Consider the language $L=\{\varepsilon,0\}$ over the alphabet $\Sigma=\{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$R=0+\varepsilon$$

**Therefore, the regular expression is** $0+\varepsilon$ **.**

Comment

**Step 12** of 14

l.

Consider the language,

$L=\{w \mid w \text{ contains an even number of 0s, or contains exactly two 1s}\}$ over the alphabet $\Sigma=\{0,1\}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$R=1^*(01^*01^*)^*+0^*10^*10^*$$

The strings accepted by the regular expression are $\varepsilon,00,11,0101,010100,...$

**Therefore, the regular expression is** $1^*(01^*01^*)^*+0^*10^*10^*$ **.**

Comments (3)

**Step 13** of 14

m.

Consider the language $L=\text{The empty set}$. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$R=\phi$$

**Therefore, the regular expression is** $\phi$ **.**

Comment

**Step 14** of 14

n.

Consider the language *L* accepts all the strings except the empty string. Let *R* be the regular expression that generates the language *L*. The regular expression is as follows:

$$\begin{aligned} R &= \Sigma^+ \\ &= (0+1)^+ \end{aligned}$$

The language accepts all the strings except $\varepsilon$.

**Therefore, the regular expression is** $(0+1)^+$ **.**

Comment

## Problem

Use the procedure described in Lemma 1.55 to convert the following regular expressions to nondeterministic finite automata.

**a.** $(0 \cup 1)^* 000 (0 \cup 1)^*$

**b.** $(((00)^*(11)) \cup 01)^*$

**c.** $\emptyset^*$

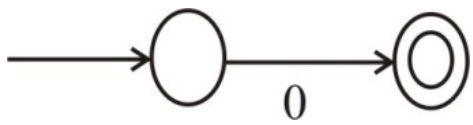
## Step-by-step solution

**Step 1** of 4

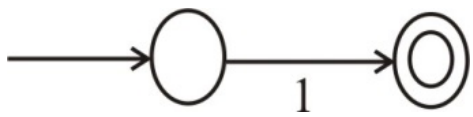
a. Consider the regular expression $R = (0 \cup 1)^* 000 (0 \cup 1)^*$.

Now, construct an NFA from this regular expression in the following procedure:

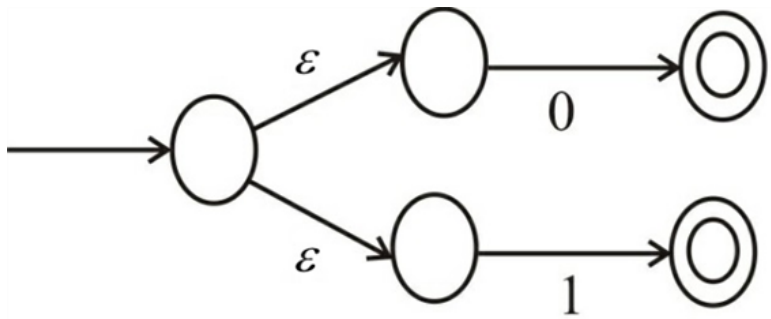
0



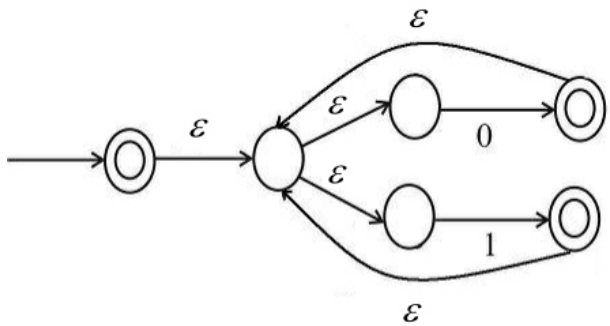

1



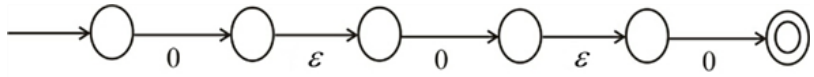

$0 \cup 1$



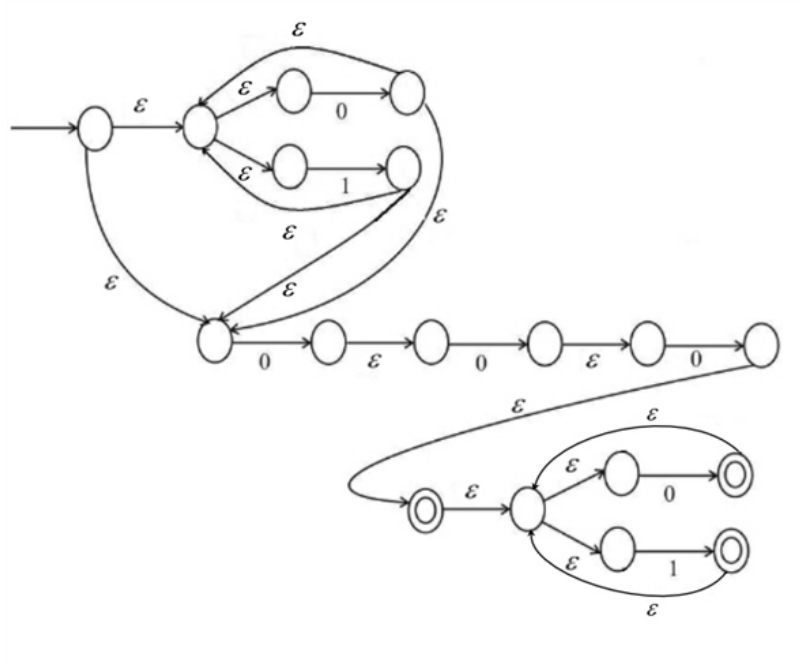

$(0 \cup 1)^*$



000




$(0\cup 1)^*000(0\cup 1)^*$



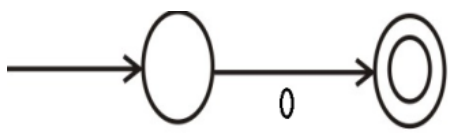

Comments (12)

**Step 2** of 4

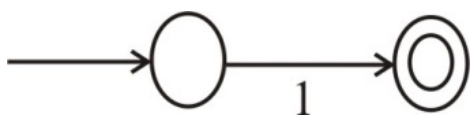
b. Consider the regular expression $R=\left(\left((00)^*(11)\right)\cup 01\right)^*$.

Now, construct the NFA from this regular expression in the following procedure:

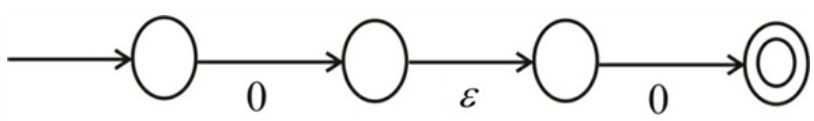
0



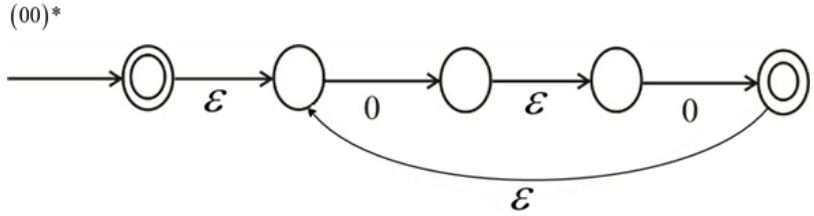

1




00




$(00)^*$

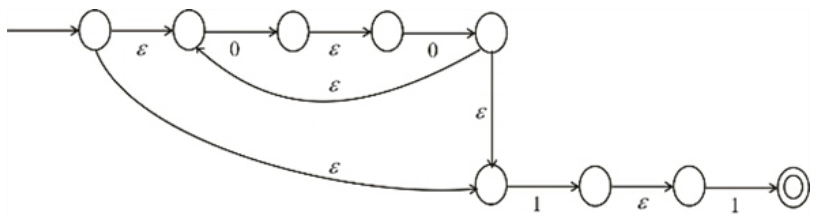



Comment

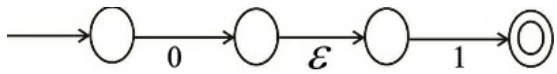
**Step 3** of 4

11

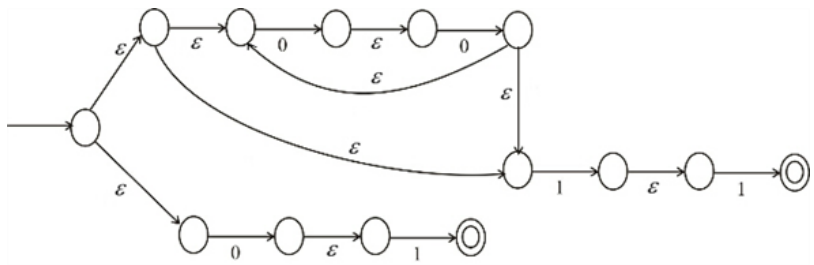
$(00)^*11$



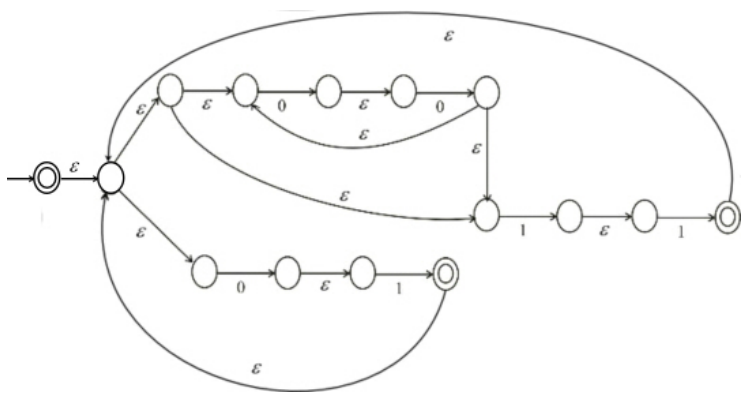

01




$(((00)^*(11))\cup(01))$




$(((00)^*(11))\cup(01))^*$




Comments (2)

**Step 4** of 4

c. Consider the regular expression $R = \phi^*$.

The closure of an empty set is an empty string i.e., $\phi^* = \{\varepsilon\}$. The NFA for the regular expression is as follows:

Comment

## Problem

For each of the following languages, give two strings that are members and two strings that are *not* members—a total of four strings for each part. Assume the alphabet $\Sigma = \{a,b\}$ in all parts.

**a.** $a^*b^*$

**b.** $a(ba)^*b$

**c.** $a^* \cup b^*$

**d.** $(aaa)^*$

**e.** $\Sigma^* a \Sigma^* b \Sigma^* a \Sigma^*$

**f.** $aba \cup bab$

**g.** $(\varepsilon \cup a)b$

**h.** $(a \cup ba \cup bb)\Sigma^*$

## Step-by-step solution

**Step 1** of 8

(a) Language is $L = a^*b^*$ over the alphabet $\Sigma = \{a,b\}$

• Strings that are member of *L*

(i) $ab$

(ii) $abb$

• Strings that are not members of *L*

(i) $ba$

(ii) $bba$

Comments (3)

**Step 2** of 8

(b) Language is $L = a(ba)^*b$ over $\Sigma = \{a,b\}$

• Strings that are members of *L*

(i) $abab$

(ii) $ababab$

• Strings that are not members of *L*

(i) $aba$

(ii) $bab$

Comments (4)

**Step 3** of 8

(c) Given language is $L = a^* \cup b^*$ over $\Sigma = \{a,b\}$

• Strings that are members of *L*

(i) $aaa$

(ii) $bbb$

• Strings that are not members of *L*

(i) $baab$

(ii) $bbaa$

Comments (4)

**Step 4** of 8

(d) Given language is $L=(aaa)^*$ over alphabet $\Sigma=\{a,b\}$

• Strings that are members of *L*

(i) $aaa$

(ii) $aaaaaa$

• Strings that not members of *L*

(i) $a$

(ii) $aaaaa$

Comments (3)

**Step 5** of 8

(e) Given language is $L=\Sigma^* a\,\Sigma^* b\Sigma^* a\Sigma^*$ over $\Sigma=\{a,b\}$

• Strings that are members of *L*

(i) $aba$

(ii) $aabbaa$

• Strings that over not members of $L$

(i) $a$

(ii) $b$

Comments (1)

**Step 6** of 8

(f) Given language is $L=aba\cup bab$ over $\Sigma=\{a,b\}$

• Strings that are members of *L*

(i) $aba$

(ii) $bab$

• Strings that over not members of $L$

(i) $abb$

(ii) $ba$

Comments (5)

**Step 7** of 8

(g) Given language is $L=(\in\cup a)b$ over $\Sigma=a\{a,b\}$

• Strings that are members of *L*

(i) $b$

(ii) $ab$

• Strings that are not members of *L*

(i) $a$

(ii) $ba$

Comments (1)

**Step 8** of 8

(h) Given language is $L = (a \cup ba \cup bb)\Sigma^*$ over $\Sigma = \{a, b\}$

• Strings that are members of *L*

(i) $a$

(ii) $bbab$

• Strings that are not members of *L*

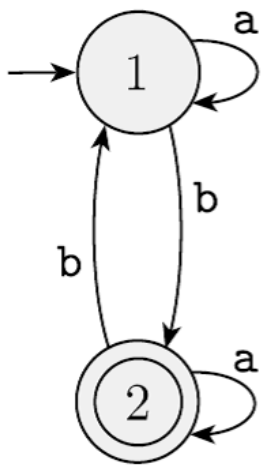
(i) $b$

(ii) $\in$

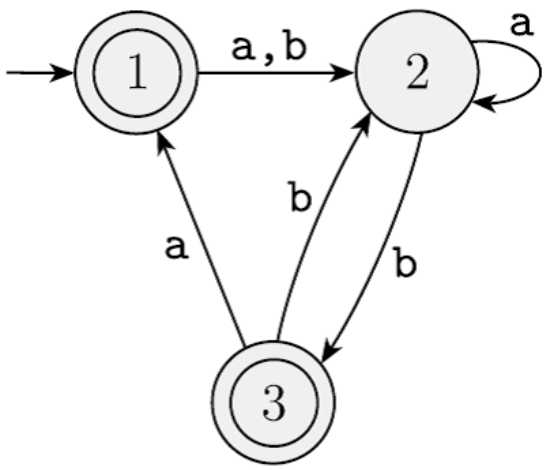
Comments (4)

## Problem

Use the procedure described in Lemma 1.60 to convert the following finite automata to regular expressions.




(a)



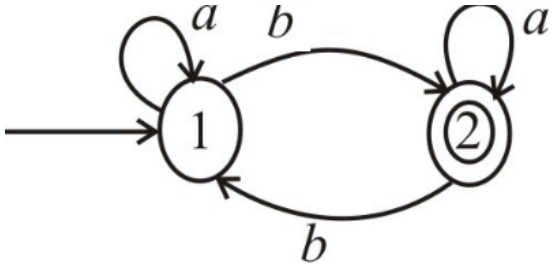

(b)

## Step-by-step solution

**Step 1** of 14

**(a)**

Consider the Finite Automata:



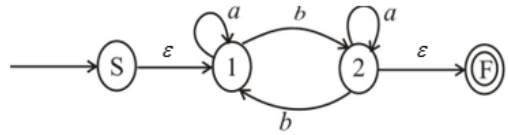

Now convert this finite automaton to a regular expression in the following steps as below:

Comment

**Step 2** of 14

**Step 1:**

Add the start state(S) and new accept state (F) to make the original accept state as non – accepting state as:

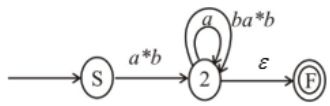



Comment

**Step 3** of 14

**Step 2:**

In the second step eliminate the state(1), no need to add the loop for the state 1 and directly add loop to the state (2) and write the expression by passing state (S) to state (2)

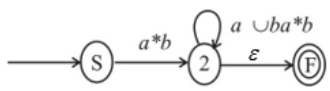



Comment

**Step 4** of 14

**Step 3:**

From the above step one loop is represented a s union with the a as follows:

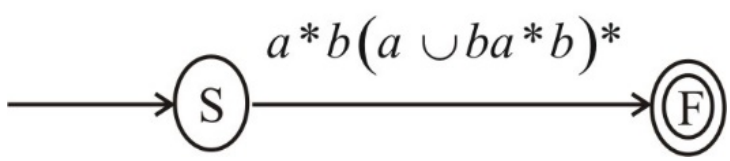



Comment

**Step 5** of 14

**Step 4:**

Now remove the loop over the state (2) and eliminate it and write expression directly from state (S) to State(F)




Comment

**Step 6** of 14

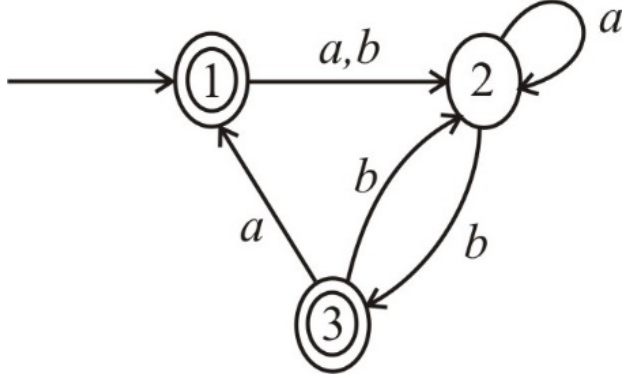
So, the regular expression for the given finite automata is $\boxed{a*b(a \cup ba*b)*}$

Comment

**Step 7** of 14

**(b)**

Consider the second finite automata is



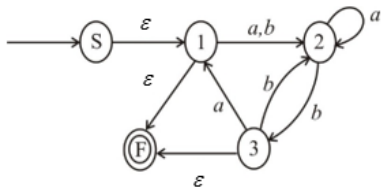

Now convert this finite automaton to a regular expression in the following steps as below:

Comment

**Step 8** of 14

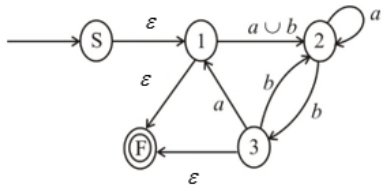
**Step 1:**

Add new start state (S) and new accept state (F). Make original accept states as non-accepting states then the Finite Automata becomes:




**Step 2:**

Perform union on the edge from state 1 to state2.

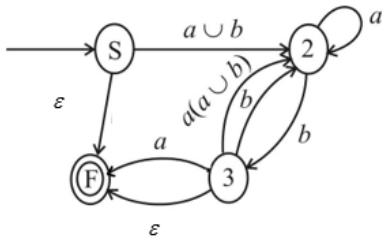



Comment

**Step 9** of 14

**Step 3:**

From the above step 2, there are no unions or loops for the state 1, So eliminate the state 1 as follows:

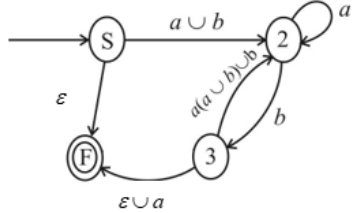



Comments (4)

**Step 10** of 14

**Step 4:**

Perform unions on edges from state 3 to state 2 and from state 3 to the final state, Then the Automata becomes as below:

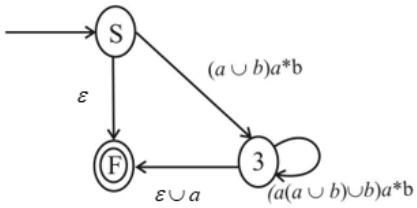



Comment

**Step 11** of 14

**Step 5:**

Here to minimize the automata eliminate 2 and perform union on 3 and write expression for the state (S) to state(3), then apply loop on state (3) with the expression of state(2)

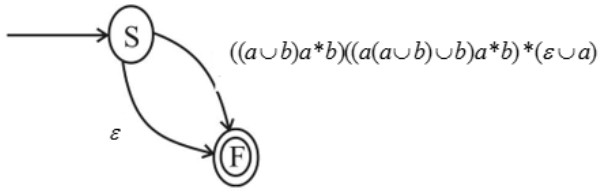



Comment

**Step 12** of 14

**Step 6:**

Eliminate the state 3 and write the expression from state(S) to state (F), because there are no loops and unions.

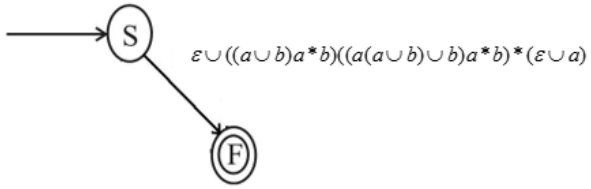



Comment

**Step 13** of 14

**Step 7:**

Perform union on edge from state *S* to state *F*




Comment

**Step 14** of 14

So, the regular expression for the given finite automata is

$$\varepsilon \cup ((a \cup b)a^*b)((a(a \cup b) \cup b)a^*b)^*(\varepsilon \cup a)$$

Comment

## Problem

In certain programming languages, comments appear between delimiters such as /# and #/. Let C be the language of all valid delimited comment strings. A member of C must begin with /# and end with #/ but have no intervening #/. For simplicity, assume that the alphabet for C is ∑ = {a, b, /, #}.

**a.** Give a DFA that recognizes C.

**b.** Give a regular expression that generates C.

## Step-by-step solution

**Step 1** of 2

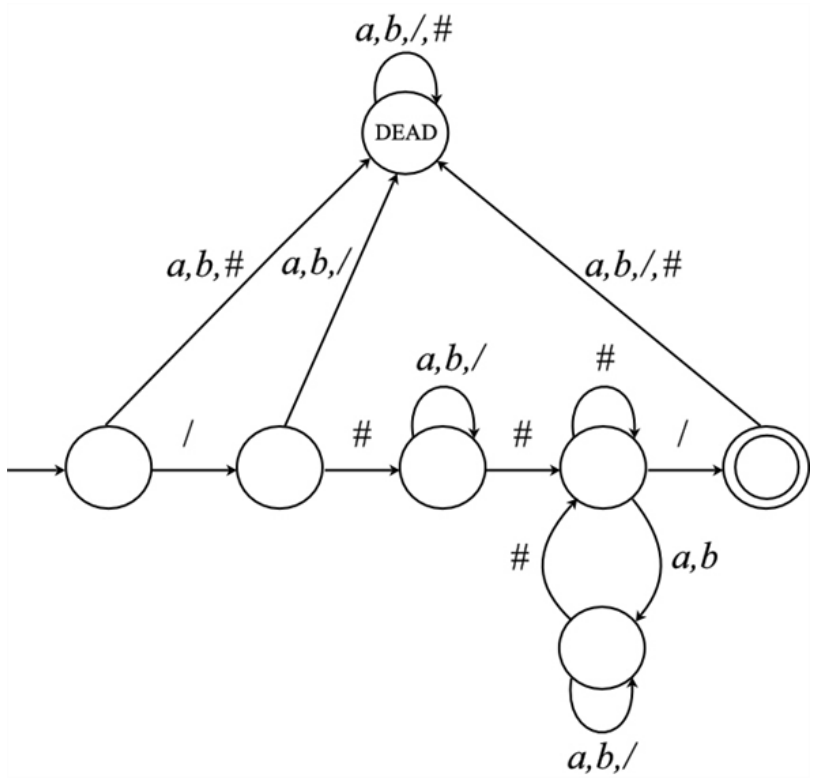
(a) Consider the language C of all valid delimited commented strings

• A member of C must begin with '/#' and end with '#/' but have no intervening '#/'.

• The alphabet of *C is* $\Sigma = \{a, b, /, \#\}$

Let *M* be the DFA that recognize the language C.

The state diagram of *M* is as follows




So this is the DFA that recognizes the languages of C.

Comments (2)

**Step 2** of 2

(b) The regular expression that generates language C is,

$/\#\left(a+b+/\right)*\#(\#+(a+b)(a+b+/)*\#)*/$

Comments (3)

## Problem

Let B be any language over the alphabet $\Sigma$. Prove that $B = B^+$ iff $BB \subseteq B$.

## Step-by-step solution

**Step 1** of 3

Let B be any language over the alphabet $\Sigma$.

**To Prove**: $B = B^+$ iff $BB \subseteq B$

The requirement is to prove both the directions of iff.

Comment

**Step 2** of 3

**One direction:**

**Assume:** $B = B^+$ .....(1)

**To show**: $BB \subseteq B$

Since, for every language $BB \subseteq B^+$ .....(2)

By substituting $(1)$ in $(2)$, it can be obtained that $BB \subseteq B$

Hence, it has been proved that: $BB \subseteq B$ iff $B = B^+$

Comment

**Step 3** of 3

**Other direction:**

**Assume**: $BB \subseteq B$

**To prove**: $B = B^+$

It is known that for every language $BB \subseteq B^+$ .....$(3)$.

Let *w* be a string of elements, such that $w \in B^+$ then $w \in B$ .

• If $w \in B^+$ then the string *w* can be split into elements $x_1, x_2, x_3 .... x_k$ such that $w = x_1, x_2 ..... x_k$ for $x_i \in B$ and $k \geq 1$. Since, $x_1, x_2 \in B$ and it is assumed that $BB \in B$.

• Similarly, it holds for the element $x_3$, such that $x_3 \in B$ and $BB \subseteq B$ .Thus, $x_1, x_2, x_3 \in B$.

• If this procedure is continued, then $x_1, x_2, x_3 .... x_k \in B$ that is $w \in B$.

• It means that, if all the elements of the string ( $x_1, x_2 .... x_k$ ) belong to B, then the strin $w \in B$.

Thus, $B^+ \subseteq B$ .....(4)

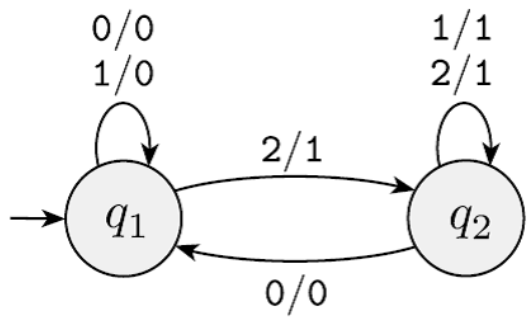
From, $(3)$ and $(4)$ it can be concluded that $B = B^+$.

Hence, it has been proved for both the directions that, for any language B, $B = B^+$ iff $BB \subseteq B$.

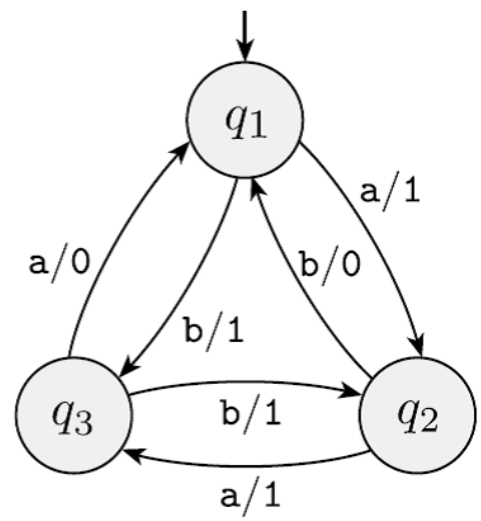
Comments (1)

## Problem

A ***finite state transducer*** (FST) is a type of deterministic finite automaton whose output is a string and not just accept or reject . The following are state diagrams of finite state transducers $T_1$ and $T_2$.




$T_1$




$T_2$

Each transition of an FST is labeled with two symbols, one designating the input symbol for that transition and the other designating the output symbol. The two symbols are written with a slash, /, separating them. In $T_1$, the transition from $q_1$ to $q_2$ has input symbol 2 and output symbol 1. Some transitions may have multiple input–output pairs, such as the transition in $T_1$ from $q_1$ to itself. When an FST computes on an input string w, it takes the input symbols $w_1 \cdots w_n$ one by one and, starting at the start state, follows the transitions by matching the input labels with the sequence of symbols $w_1 \cdots w_n = w$. Every time it goes along a transition, it outputs the corresponding output symbol. For example, on input 2212011, machine $T_1$ enters the sequence of states $q_1, q_2, q_2, q_2, q_2, q_1, q_1, q_1$ and produces output 1111000. On input abbb, $T_2$ outputs 1011. Give the sequence of states entered and the output produced in each of the following parts.

**a.** $T_1$ on input 011

**b.** $T_1$ on input 211

**c.** $T_1$ on input 121

**d.** $T_1$ on input 0202

**e.** $T_2$ on input b

**f.** $T_2$ on input bbab

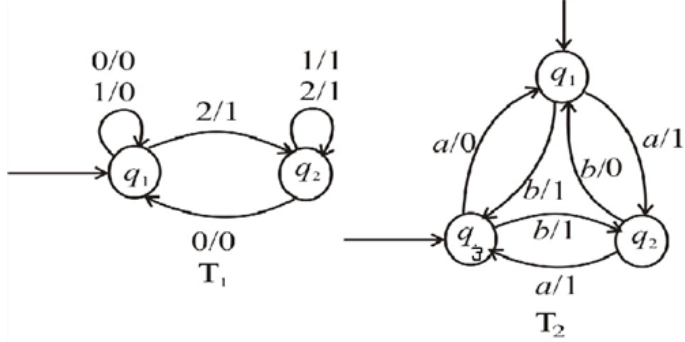
**g.** $T_2$ on input bbbbbb

**h.** $T_2$ on input $\varepsilon$

## Step-by-step solution

**Step 1** of 9

Given state diagram of finite state transducers (FST) $T_1$ and $T_2$ are as follows




Comment

**Step 2** of 9

(a) Input 011

FST $T_1$ will enter the following sequence of states on input 011: $q_1, q_1, q_1, q_1$ and produces the output: 000

Comment

**Step 3** of 9

(b) Input $211$

FST $T_1$ will enter the following sequence of state on input 211: $q_1, q_2, q_2, q_2$ and produces the output: 111

Comment

**Step 4** of 9

(c) Input 121

FST $T_1$ will enter the following sequence of states on input $121$: $q_1, q_1, q_2, q_2$ and produces the output 011

Comment

**Step 5** of 9

(d) Input 0202

FST $T_1$ will enter the following sequence of states on input 0202: $q_1, q_1, q_2, q_1, q_2$

and produces the output: 0101

Comment

**Step 6** of 9

(e) Input *b*

FST $T_2$ will enter the following sequence of states on input *b*: $q_1, q_3$ and produces the output: 1

Comment

**Step 7** of 9

(f) Input $bbab$

FST $T_2$ will enter the following sequence of states on input $bbab$: $q_1, q_3, q_2, q_3, q_2$

and produces the output 1111

Comment

**Step 8** of 9

(g) Input $bbbbbb$

FST $T_2$ will enter the following sequene of states on input $bbbbbb$:

$q_1, q_3, q_2, q_1, q_3, q_2, q_1$ and produces the output: 110110

Comment

**Step 9** of 9

(h) Input $\in$

FST $T_2$ will enter the following sequence of states on input $\in$: $q_1$

and produces the following output $\in$

Comment

## Problem

Read the informal definition of the finite state transducer given in Exercise 1.24. Give a formal definition of this model, following the pattern in Definition 1.5 (page 35). Assume that an FST has an input alphabet ∑ and an output alphabet $\Gamma$ but not a set of accept states. Include a formal definition of the computation of an FST.

(Hint: An FST is a 5-tuple. Its transition function is of the form $\delta: Q \times \Sigma \longrightarrow Q \times \Gamma.)$

## Step-by-step solution

**Step 1** of 2

**Finite State Transducer:**

• A finite state transducer is a kind of deterministic finite state automaton which consists of both the input string and the output string.

• It converts the input string into an output string.

Comment

**Step 2** of 2

**The formal definition of Finite State Transducer (FST) is as follows:**

A Finite State Transducer is a 6-tuple machine and is represented as $M = (Q, \Sigma, \Gamma, \delta, q_0)$ where

- $Q$ is a finite set of states.
- $\Sigma$ is a finite set of input alphabets.
- $\Gamma$ is a finite set of output alphabets.
- $\delta : Q \times \Sigma \to Q \times \Gamma$ is the transition function that defines rules.
- $q_0 \in Q$, is the start state.

**The formal definition of the computation of Finite State Transducer (FST) is as follows:**

• The computation of the finite state machine is carried out by translating the input string into output string.

• Assume that *w* is an input string over the input alphabet $\Sigma$ and *x* is an output string consisting of alphabet of $\Gamma$.

• The transition is carried out over a sequence of states $q_0', q_1', \ldots, q_n'$ in $Q'$ such that

- $q_0' = q_0$

• The transition of $\delta(q_{i+1}', w_{i+1}) = (q_{i+1}', x_{i+1})$ for $0 \le i < n$.

Comments (1)

## Problem

Using the solution you gave to Exercise 1.25, give a formal description of the machines $T_1$ and $T_2$ depicted in Exercise 1.24

## Step-by-step solution

**Step 1** of 3

Here, the formal description of Turing machine $T_1$ and $T_2$ need to be defined. A finite state transducer (FST) is formally defined by the $(Q, \Sigma, \Gamma, \delta, q_0)$ tuple, where:

- The finite set of states is $Q$.
- The input alphabet is $\Sigma$.
- The output alphabet is $\Gamma$.
- The transition function $\delta$ takes a state and an input symbol and returns a state and an output symbol.

$\delta : Q \times \Sigma \to Q \times \Gamma$

- The start state $q_0$.

Comment

**Step 2** of 3

The finite state transducer $T_1$ is formally defined by the $(\{q_1, q_2\}, \{0,1,2\}, \{0,1\}, \delta_1, q_1)$, where the transition function $\delta_1$ is as follows:

| *Input* / *State* | 0 | 1 | 2 |
|---|---|---|---|
| $q_1$ | $\{q_1, 0\}$ | $\{q_1, 0\}$ | $\{q_2, 1\}$ |
| $q_2$ | $\{q_1, 0\}$ | $\{q_2, 1\}$ | $\{q_2, 1\}$ |

Comment

**Step 3** of 3

The second FST is defined as $T_2 = (\{q_1, q_2, q_3\}, \{a, b\}, \{0,1\}, \delta_2, q_1)$. The transition function $\delta_2$ is given by:

| *Input* / *State* | $a$ | $b$ |
|---|---|---|
| $q_1$ | $\{q_2, 1\}$ | $\{q_3, 1\}$ |
| $q_2$ | $\{q_3, 1\}$ | $\{q_1, 0\}$ |
| $q_3$ | $\{q_1, 0\}$ | $\{q_2, 1\}$ |

Comment

## Problem

Read the informal definition of the finite state transducer given in Exercise 1.24. Give the state diagram of an FST with the following behavior. Its input and output alphabets are {0,1}. Its output string is identical to the input string on the even positions but inverted on the odd positions. For example, on input 0000111 it should output 1010010.

## Step-by-step solution

**Step 1** of 2

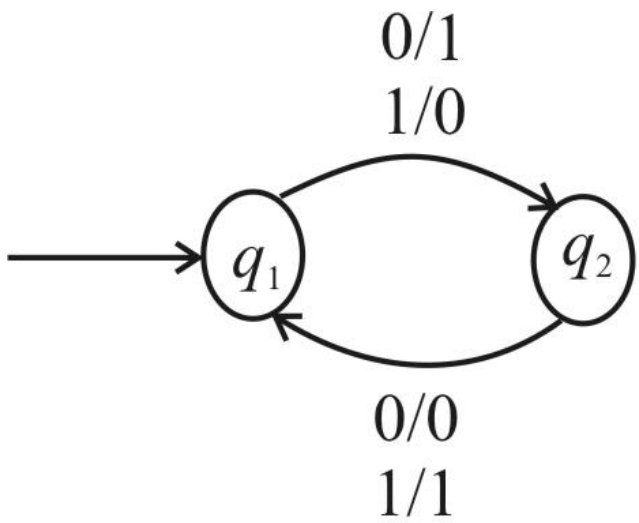
Given input and output alphabets for FST are $\{0,1\}$.

FST has the following behavior

• Its output string is identical to the input string on the even positions.

• Its output string is inverted on the odd positions.

So the state diagram of that FST is given as follows:




Comment

**Step 2** of 2

So by this state diagram,

• In even positions output string is same as input string

i.e. for input $0 \rightarrow$ output is also 0

for input $1 \rightarrow$ output is also 1

• In odd positions, out string is inverted

i.e. for input $0 \rightarrow$ output is 1

for input $1 \rightarrow$ output is 0.

Comment

## Problem

Convert the following regular expressions to NFAs using the procedure given in Theorem 1.54. In all parts, ∑ = {a, b}.

**a.** $a(abb)^* \cup b$

**b.** $a^+ \cup (ab)^+$

**c.** $(a \cup b^+)a^+b^+$

**THEOREM 1.54**

A language is regular if and only if some regular expression describes it.

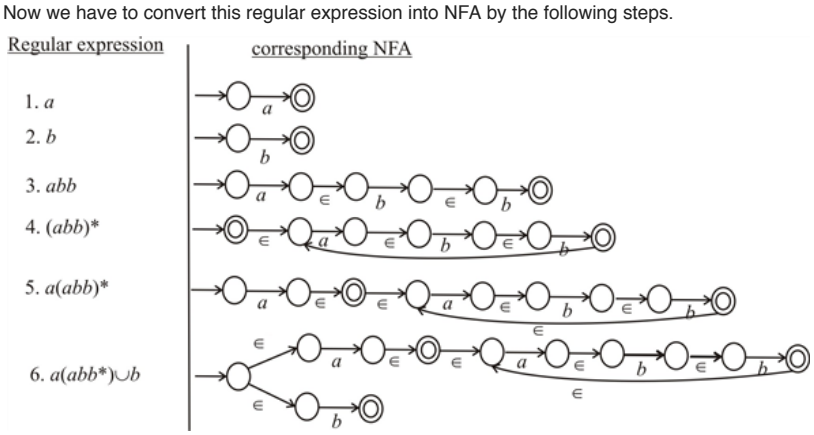
## Step-by-step solution

**Step 1** of 3

(a) Given regular expression

$R = a(abb)^* \cup b$ over $\Sigma = \{a, b\}$.

Now we have to convert this regular expression into NFA by the following steps.



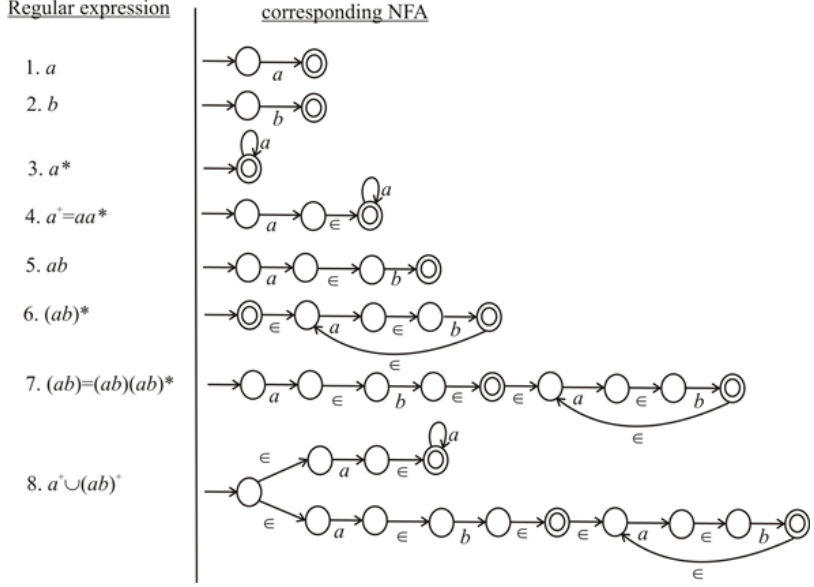

Comments (4)

**Step 2** of 3

(b) Given regular expression is

$R = a^+ \cup (ab)^+$ over $\Sigma = \{a, b\}$

Now we have to convert this regular expression into NFA by the following steps.

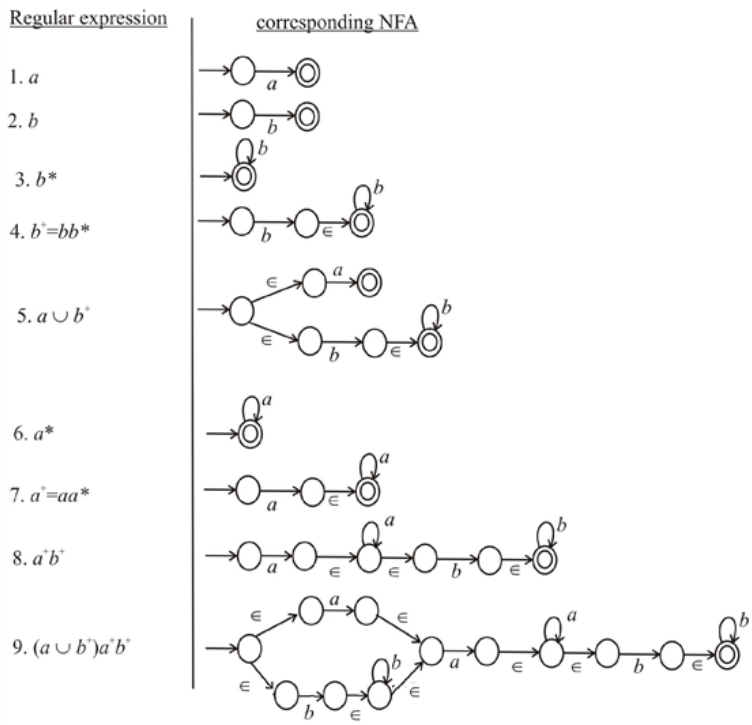



Comment

**Step 3** of 3

(c) Given regular expression is $R=\left(a\cup b^{+}\right)a^{+}b^{+}$ over $\Sigma=\{a,b\}$.

Now we have to convert this regular expression R into NFA by the following steps.




Comments (2)

## Problem

Use the pumping lemma to show that the following languages are not regular.

A **a.** $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$

**b.** $A_2 = \{www \mid w \in \{a, b\}^*\}$

A **c.** $A_3 = \{a^{2^n} \mid n \geq 0\}$ (Here, $a^{2^n}$ means a string of $2^n$ a's.)

## Step-by-step solution

**Step 1** of 4

Pumping Lemma:

If A is regular language, there is a number $p$ ( the pumping length) where *S* is any string in A of length at least *P*, then *S* may be divided into three pieces, $S = xyz$, satisfying the following conditions.

1. For each $i \geq 0, xy^i z \in A$
2. $|y| > 0$, and
3. $|xy| \leq p$

Comment

**Step 2** of 4

(a)

Consider the language, $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$.

Assume $A_1$ is a regular language.

Let $p$ be the pumping length given by the pumping lemma consider a string $S = 0^p 1^p 2^p \in A_1$

$|S| > P$ so, by pumping lemma, take $S = 0^p 1^p 2^p = xyz$ such that $|xy| \leq p, |y| > 0$ consider the following 2 possibilities:

Let 001122 be the string that belongs to $A_1$. $S = 0^p 1^p 2^p = 001122$. The pumping length of the string is 2. To satisfy the conditions of the pumping lemma, x = 0, y = 0, z = 1122.

$$\begin{aligned} S &= 001122 \\ &= \underset{x}{0}\ \underset{y}{0}\ \underset{z}{1122} \end{aligned}$$

Pump the middle part such that $xy^i z\ (i \geq 0)$. For *i*=2, the *y* becomes 00. The string after pumping is 0001122.

$$\begin{aligned} S &= (0)\ (0)^i\ (1122) \\ &= \underset{x}{0}\ \underset{y}{00}\ \underset{z}{1122} \quad [when\ i = 2] \end{aligned}$$

The string $0001122 \notin A_1$ because the string that is accepted by the language should have equal number of 0's, 1's and 2's. It is a contradiction. So, the pumping lemma is violated.

**Therefore, $A_1$ is not a regular language.**

Comment

**Step 3** of 4

(b)

Consider the language, $A_2 = \{www \mid w \in \{a,b\}^*\}$.

Assume $A_2$ is a regular language.

Let $p$ be the pumping length given by the pumping lemma.

Consider a string $S = a^p b a^p b a^p b \in A_2$.

By pumping lemma, this string can be divided into three pieces $xyz$ such that $|xy| \le p, |y| > 0$ and $xy^i z \in A_2 \forall i \ge 0$

So $S = a^p b a^p b a^p b = xyz$.

Let *aabaabaab* be the string that belongs to $A_2$. The pumping length of the string is 2. To satisfy the conditions of the pumping lemma, x = *a*, y = *a*, z = *baabaab*.

$$S = aabaabaab$$
$$= \underset{x}{a}\ \underset{y}{a}\ \underset{z}{baabaab}$$

Pump the middle part such that $xy^i z\ (i \ge 0)$. For *i*=2, the *y* becomes *aa*. The string after pumping is *aaabaabaab*.

$$S = (a)\ (a)^i\ (baabaab)$$
$$= \underset{x}{a}\ \underset{y}{aa}\ \underset{z}{baabaab} \quad [when\ i = 2]$$

The string $aaabaabaab \notin A_2$.It is a contradiction. So, the pumping lemma is violated.

**Therefore, $A_2$ is not a regular language.**

Comments (8)

**Step 4** of 4

(c)

Consider the language, $A_3 = \left\{a^{2^n} \mid n \ge 0\right\}$ (Here, $a^{2^n}$ means a string of $2^n$ a's).

Assume that $A_3$ is regular language.

Let $p$ be the pumping length given by pumping lemma consider a string $S = a^{2^p} \in A_3$. And $|S| > p$

By pumping lemma, this string can be divided into three pieces $xyz$ such that $|xy| \le p, |y| > 0$ and $xy^i z \in A_2 \forall i \ge 0$

Let *aaaa* be the string that belongs to $A_3$. The pumping length of the string is 2. To satisfy the conditions of the pumping lemma, x = *a*, y = *a*, z = *aa*.

$$S = aaaa$$
$$= \underset{x}{a}\ \underset{y}{a}\ \underset{z}{aa}$$

Pump the middle part such that $xy^i z\ (i \ge 0)$. For *i*=2, the *y* becomes *aa*. The string after pumping is *aaaaa*.

$$S = (a)\ (a)^i\ (aa)$$
$$= \underset{x}{a}\ \underset{y}{aa}\ \underset{z}{aa} \quad [when\ i = 2]$$

The string $aaaaa \notin A_3$.It is a contradiction. So, the pumping lemma is violated.

**Therefore, $A_3$ is not a regular language.**

Comments (9)

## Problem

Describe the error in the following "proof" that $0^*1^*$ is not a regular language. (An error must exist because $0^*1^*$ *is* regular.) The proof is by contradiction. Assume that $0^*1^*$ is regular. Let p be the pumping length for $0^*1^*$ given by the pumping lemma. Choose s to be the string $0^p1^p$. You know that s is a member of $0^*1^*$, but Example 1.73 shows that s cannot be pumped. Thus you have a contradiction. So $0^*1^*$ is not regular.

## Step-by-step solution

**Step 1** of 4

The pumping lemma is used as a negative test to prove that the given language is non-regular. The language that violates the any of the three conditions of the pumping lemma is classified as non-regular.

Since the pumping lemma starts by assuming that the given language is regular, the belongingness of the string, which is used as a counter example, is tested only for the given language and not for all the languages that accepts the string.

Comment

**Step 2** of 4

The proof given in the example 1.73 (refer to the textbook example 1.73) is for a different language. The counter examples given to prove the language as non-regular does not hold true for the language given in the question as shown below:

Let A be the language $\{0^*1^*\}$ given in the question.

Let B be the language $\{0^n1^n \mid n>=0\}$ given in the example 1.73.

As per the example 1.73, s is $0^p1^p$ and hence as per the pumping lemma, s should be split into three pieces as s = xyz, where for any i>=0, the string $xy^iz$ is in B. The cases are as follows:

• The string y contains all 0s: The string xyyz will result in more number of 0s than the number of 1s which does not belong to the language B but the string belongs to language A and hence, the pumping lemma is not violated.

• The same reason holds true for the case when the string y contains all 1s. The string xyyz will results in more number of 1s which is again accept by the given language A.

Comment

**Step 3** of 4

Since the above conditions are satisfied, the given language cannot be classified as non-regular by pumping lemma.

Comment

**Step 4** of 4

Therefore, the error in the given proof is that a string which is used as a counter example to prove a certain language as non-regular does not signifies that all the languages that accept that string are considered as non-regular.

The above problems arise when the pumping lemma is used on a language which is regular since violating a condition of the pumping lemma indicates that the language is non-regular but the vice versa is not always true.

Comment

## Problem

For any string $w = w_1 w_2 \cdots w_n$, the ***reverse*** of w, written $w^R$, is the string w in reverse order, $w_n \cdots w_2 w_1$. For any language A, let $A^R = \{w^R | w \in A\}$. Show that if A is regular, so is $A^R$.

## Step-by-step solution

**Step 1** of 3

Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA that recognizes *A*,

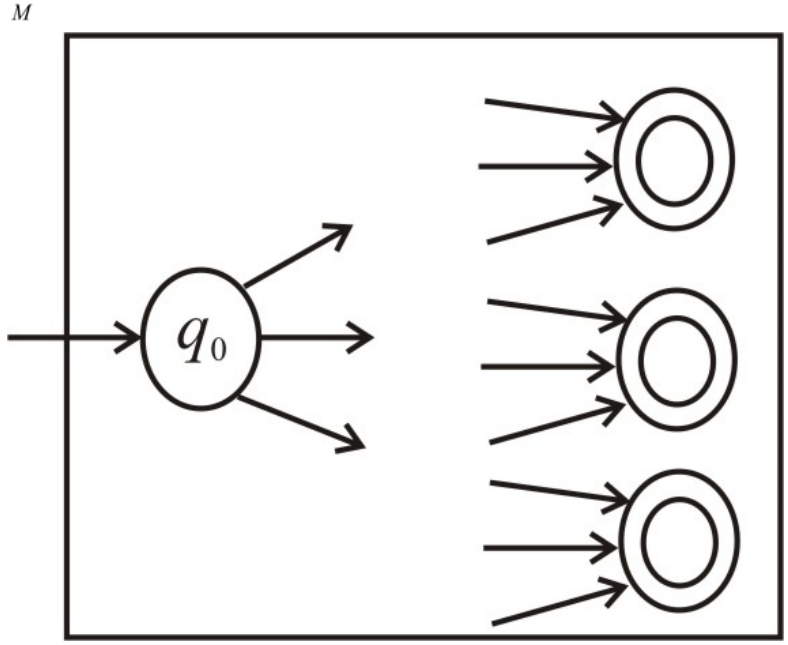
Now we build a NFA $M'$ for $A^R$ as follows:

• Reverse all the arrows of *M*

• Convert the start state for M as the only accept state $q'_{accept}$ for $M'$.

• Add a new start state $q_0'$ for $M'$, and from $q_0'$, add $\in -$transitions to each state of $M'$ corresponding to accept states of *M*.

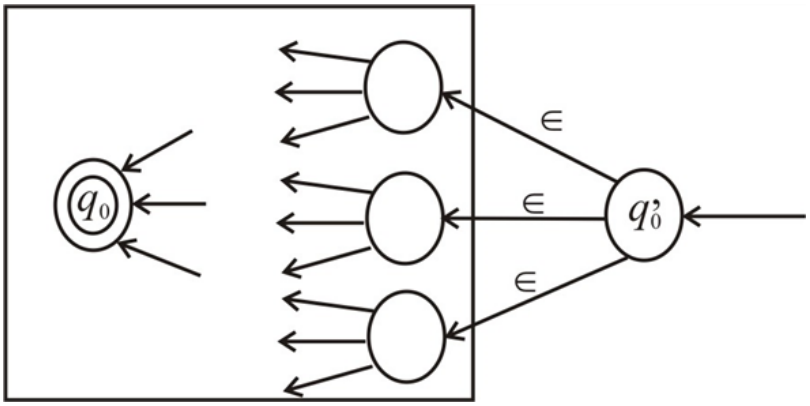
$M$




Comment

**Step 2** of 3

$M'$:




Comment

**Step 3** of 3

Here $q'_0 = q'_{accept}$

• For any $w \in \Sigma *$, there is a path following *w* from the start state to an accept sate in *M* iff there is a path following $w^R$ from $q'_0$ to $q'_{\text{accept}}$ in $M'$

• That means that $w \in A$ iff $w^R \in A^R$.

Comments (1)

**Problem**

$$\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

$\Sigma_3$ contains all size 3 columns of 0s and 1s. A string of symbols in $\Sigma_3$ gives three rows of 0s and 1s. Consider each row to be a binary number and let

$$B = \{w \in \Sigma_3^* \mid \text{the bottom row of } w \text{ is the sum of the top two rows}\}.$$

For example,

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \in B, \quad \text{but} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \notin B.$$

Show that $B$ is regular. (Hint: Working with $B^{\mathcal{R}}$ is easier. You may assume the result claimed in Problem 1.31.)

**Step-by-step solution**

**Step 1** of 3

**Consider the data**

- $\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \ldots \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}$
- **A string of symbols in $\Sigma_3$ gives 3 rows of $0s$ and $1s$.**
- **Each row to be a binary number**
- $B = \{w \in \Sigma_3^* \text{ the bottom row of w is the same of the top two row}\}$ **is the language over $\Sigma_3$.**

Comment

**Step 2** of 3

**Already know that "regular languages are closed under reversal".**
**Then, if prove that $B^R$ is regular, then automatically $B$ is regular and vice – versa.**
**So, first have to prove that $B^R$ is regular.**
**A language is said to be regular if some automaton recognizes it.**

Comment

**Step 3** of 3

Let M be the automaton that recognizes $B^R$.

- $M$ has 2 states.

(i) $c_0$, which denotes that the string that we have read so for leads to a carry 0.

(ii) $c_1$, that stands for carry 1.

Now $M = (Q, \Sigma, \delta, q_0, F)$

Where $Q = \{c_0, c_1\}$

= set of states

$$\Sigma = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \ldots \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}$$

= set of alphabets

$q_0 = c_0$

= start state

$F = \{c_0\}$

= set of final states.

$\delta$ is given as:

- $\delta(c_0, a) = c_0$ if $a = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$ or $\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$
- $\delta(c_0, a) = c_1$ if $a = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$
- $\delta(c_1, a) = c_1$ if $a = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$
- $\delta(c_1, a) = c_0$ if $a = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

All other arrows go to trap state. Then, the defined a automation $M$ to recognize $B^R$

**Therefore $B^R$ is a regular language. As $B^R$ is regular, $B$ is also a regular language.**

Comment

## Problem

Let

$$\Sigma_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

Here, $\Sigma_2$ contains all columns of 0s and 1s of height two. A string of symbols in $\Sigma_2$ gives two rows of 0s and 1s. Consider each row to be a binary number and let

$$C = \{w \in \Sigma_2^* | \text{ the bottom row of } w \text{ is three times the top row}\}.$$

For example, $\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} \in C$, but $\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} \notin C$. Show that $C$ is regular. (You may assume the result claimed in Problem 1.31.)

## Step-by-step solution

**Step 1** of 5

Consider the language,

$C = \left\{ w \in \Sigma_2^* \mid \text{ the bottom row of } w \text{ is three times the top row} \right\}$

Over the alphabet $\Sigma_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$

Comment

**Step 2** of 5

Here each row is binary number.

- The regular languages are closed under reversal. Use this property to prove the language *C* is regular.
- Scan the input in reverse order.
- Begin with the lower order bits and multiply it by 3 in binary format to add the multiplicand with the result of shifting the multiplicand itself by 1 bit.
- Consider a binary number 110 (6). Three times of 110(6) is 18 (10010) which is obtained by adding 110(6) with 1100(12).
- The top row can be represented as $X_n X_{n-1} ... X_1 X_0$.
- The bottom row can be represented as $P_n P_{n-1} ... P_1 P_0$.
- Add $X_{n-1} X_{n-2} ... X_0 \; 0$ to the top row to obtain the bottom row. It can be represented as follows:

$$\begin{array}{l} X_n X_{n-1} ..... X_1 X_0 \\ X_{n-1} X_{n-2} ... X_0 \; 0 \\ \hline P_n P_{n-1} ..... P_1 P_0 \end{array}$$

- The value of $P_i$ depends on the values of $X_i$ and $X_{i-1}$.
- The bit at the top of the column is the carry-in ( $C_i^{in}$). For the first column the carry-in will be zero.
- If two 1 bits are added, the carry will be generated. It is called carry-out ( $C_{i-1}^{out}$ ).
- The $P_i$ can be obtained by performing the XOR operation on $X_i, X_{i-1}$ and $C_i^{in}$. The carry-out of the currently working column will be given as the carry-in for the next column.

Comment

**Step 3** of 5

The language is said to be regular if there exists a finite automaton (DFA or NFA) for that language.

• To prove that the language C is regular, construct a DFA.

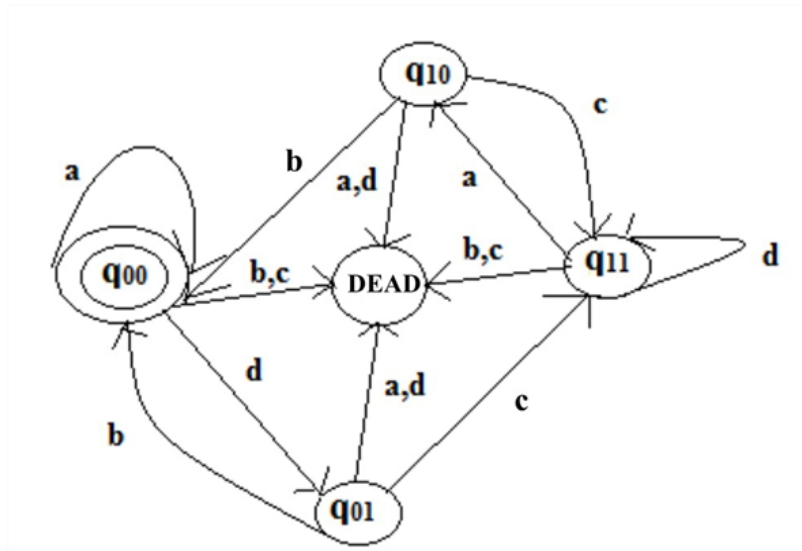
• It is necessary with 4 states without counting sink state to keep track of all the possible combinations of $C_{i+1}^{in}$ $and$ $X_{i-1}$ $and$ for each of these states, there will be exactly two possible valid output transitions depending on whether $X_i$ $is$ $0$ or $1$.

• Assume that $q_{ij}$ represents the state for which $C^{in}(Carry-in)$ is equal to *i* and the preceding symbol is observed at the top is equal to *j*.

Comment

**Step 4** of 5

The state diagram of DFA is as follows:




Where $a=\begin{bmatrix}0\\0\end{bmatrix}, b=\begin{bmatrix}0\\1\end{bmatrix}, c=\begin{bmatrix}1\\0\end{bmatrix}, d=\begin{bmatrix}1\\1\end{bmatrix}$

Consider an example string $\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}$.

Read the string in reverse order. Here, the initial state is $q00$. The input $\begin{bmatrix}0\\0\end{bmatrix}$ to the $q00$, stays in the same state. The input $\begin{bmatrix}1\\1\end{bmatrix}$ to $q00$, moves to the state $q01$. The input $\begin{bmatrix}0\\1\end{bmatrix}$ to $q01$, moves to the state $q00$. The input $\begin{bmatrix}0\\0\end{bmatrix}$ to the $q00$, stays in the same state. The string $\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}$ is accepted in reverse order.

Comment

**Step 5** of 5

Since a DFA that recognizes $C^R$ is built. Therefore $C^R$ is regular. Hence, it is proved that *C* is regular.

Comment

## Problem

Let $\Sigma_2$ be the same as in Problem 1.33. Consider each row to be a binary number and let

$$D = \{w \in \Sigma_2^* \mid \text{the top row of } w \text{ is a larger number than is the bottom row}\}.$$

For example, $\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right] \in D$, but $\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right] \notin D$. Show that $D$ is regular.

## Step-by-step solution

**Step 1** of 4

Given language is

$D = \left\{ w \in \Sigma_2^* \mid \text{ the top row of } w \text{ is the larger number than is the bottom row} \right\}$

Over the alphabet $\Sigma_2 = \left\{ \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix} \right\}$

Language for given expression L = $\left\{ \varepsilon, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}, \ldots \right\}$

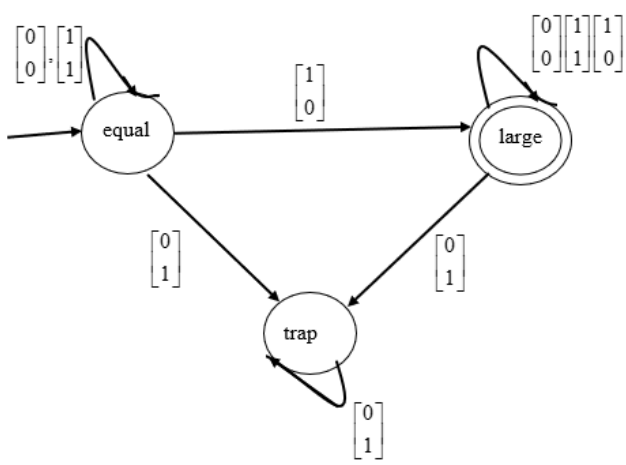
Here each row is binary number.

Comment

**Step 2** of 4

Let **M** be the DFA, over the input alphabet $\Sigma_2 = \left\{ \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix} \right\}$.

The state transition diagram of **M** is as follows:




Comments (1)

**Step 3** of 4

We must prove that *D* is a regular language.

A language is said to be regular if it recognizes by a DFA.

Let take string form language D, $w = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}$

Initial state of the above DFA is 'equal'

Parse string

$$\text{equal} \xrightarrow{\begin{bmatrix} 1 \\ 0 \end{bmatrix}} \text{large} \xrightarrow{\begin{bmatrix} 1 \\ 0 \end{bmatrix}} \text{large}$$

Here 'large' is final state , the string is accepted by the DFA.

Comment

**Step 4** of 4

Thus, language of given D is accepted by the given DFA.

we defined a DFA to recognize the language D.

Therefore, $\boxed{\text{D is a regular language.}}$

Comment

## Problem

Let $\Sigma_2$ be the same as in Problem 1.33. Consider the top and bottom rows to be strings of 0s and 1s, and let

$$E = \{w \in \Sigma_2^* \mid \text{the bottom row of } w \text{ is the reverse of the top row of } w\}.$$

Show that $E$ is not regular.

## Step-by-step solution

**Step 1** of 1

**Expression is not regular**

Consider that $\Sigma_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$

*Proof:* The bottom row of the string is the reverse of the top row of the string.

In order to prove that bottom row of string is the reverse of the top row of the string user need to prove this with the help of contradiction.

First it is to be assumed that the expression *E* is a regular language.

Suppose, the constant variable *p* is associated with expression *E*

Choose the string *s*. This is because $s \in L$

$$s = \begin{bmatrix} 1 \\ 0 \end{bmatrix}^p \begin{bmatrix} 0 \\ 1 \end{bmatrix}^p$$

$$s = 1^p 0^p$$
$$= \left(0^p 1^p\right)^R, \text{ and}$$

$$|s| = 2p \geq p$$

Now, the string *s* is being partitioned into three pieces such that,

$s = xyz$ , where $p \geq q$

$y = \begin{bmatrix} 1 \\ 0 \end{bmatrix}^p$, and $z = \begin{bmatrix} 0 \\ 1 \end{bmatrix}^p$

For any division of $y = uvw$, the value of $v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}^m$ but the condition is that $0 < m \leq p$

For any value of $i \geq 0$, suppose value of *i* is assumed to be 2.

$$s = xyz$$

Now, putting the value of *y* here,

$$s = xuv^i wz \quad s = xuv^2 wz$$

$$s = \begin{bmatrix} 1 \\ 0 \end{bmatrix}^{p+m} \begin{bmatrix} 0 \\ 1 \end{bmatrix}^p$$

Because $m > 0$ and $1^{p+m} 0^p \neq \left(0^{p+m} 1^p\right)^R$

By the above contradiction, it is being proved that the top row of the string is not the reverse of the bottom row of the string.

This implies, $xuv^2 wz \notin L$

Thus, by the pumping lemma of proof by contradiction, it is being proved that the given language *L* is not a regular language.

Comment

## Problem

Let $B_n = \{\mathtt{a}^k \mid k \text{ is a multiple of } n\}$. Show that for each $n \geq 1$, the language $B_n$ is regular.

## Step-by-step solution

**Step 1** of 5

Consider that:

$$B_n = \{a^k \mid k \text{ is a multiple of } n\}$$

In order to prove that the given expression is regular, the value of *n* is chosen as greater than or equal to 1.

Comment

**Step 2** of 5

Suppose, $k = ni$, where *i* is any positive integer. In starting, suppose value of *i* is chosen to be 1.

When $i = 1$ and $n = 1$

$$\begin{aligned} B_1 &= \{a^k\} \\ &= \{a^{ni}\} \\ &= \{a^{1\times 1}\} \\ &= \{a\} \end{aligned}$$

Then, the String comes out to be $\{a\}$

Comment

**Step 3** of 5

Increase the value of *n* keeping the value of *i* equal to 1. When $i = 1$ and $n = 2$

$$\begin{aligned} B_2 &= \{a^k\} \\ &= \{a^{ni}\} \\ &= \{a^{2\times 1}\} \\ &= \{aa\} \end{aligned}$$

Then, the String comes out to be $\{aa\}$

Comment

**Step 4** of 5

Increase the value of *n* keeping the value of *i* equal to 1. When $i = 1$ and $n = 3$

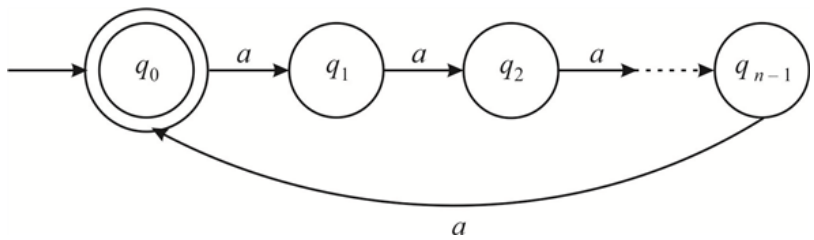
$$\begin{aligned} B_3 &= \{a^k\} \\ &= \{a^{ni}\} \\ &= \{a^{3\times 1}\} \\ &= \{aaa\} \end{aligned}$$

Then, the String comes out to be $\{aaa\}$ and so on.

Comment

**Step 5** of 5

Finite automation of the regular expression is as shown:




In the above finite automaton, $q_o$ is the initial and final state and $q_1$, $q_2$, $q_3$ and $q_{n-1}$ are the subsequent states.

The language $B_n$ is the regular language. According to the closure property of the regular expression, it is clearly seen that the specific expression is a regular expression when the value of *n* is greater than and equal to 1.

Closure property includes various operations such as union, intersection, set complement, set reversal, set difference and many more. Assume that $B_n$ is regular.

• Union of $B_1$ and $B_2$ results in the third string and it is also a regular expression.

• Similarly, if user applies any property of closure, then the result is the regular expression.

**Hence, it is proved that the above expression is the regular expression.**

Comments (2)

## Problem

Let $C_n$ = {xl x is a binary number that is a multiple of n}. Show that for each n ≥ 1, the language $C_n$ is regular

## Step-by-step solution

**Step 1** of 3

Given language is

$$C_n = \{x \mid x \text{ is a binary number that is a multiple of } n\}$$

• A language is regular if it is recognized by a DFA.

• So construct a DFA to keep track of the remainder of the input when divided by *n*.

Comment

**Step 2** of 3

Construction of DFA is as follows:

• Assume the value of n as 3.

• To determine whether a binary number is a multiple of 3, it is necessary to find its remainder modulo 3.

• If it ends up with remainder zero then accept.

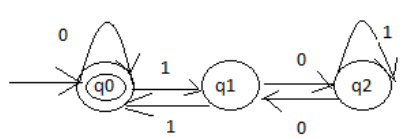
• Otherwise reject.

• Every time read a digit, the preceding string is shifted left one position thereby doubling its value x.

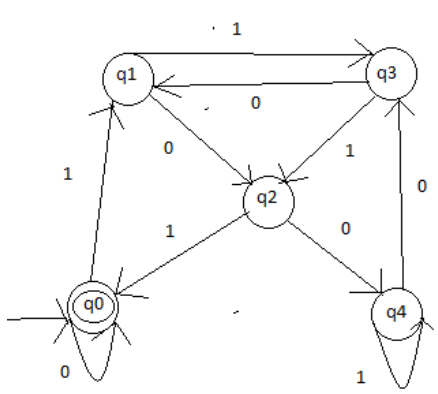
• If the current digit is 0, them the new value is $2x \pmod 3$.

• If the current digit is 1, them the new value is $2x+1 \pmod 3$.

• The DFA for the above example is as follows:




• Similarly for n=5 the DFA is as follows:



Comment

**Step 3** of 3

The other case can be proved similarly.

For example:

$C_6 = C_3 0$
$C_{12} = C_6 00$
$C_{15} = C_3 \cap C_5$
....

Thus $M = \left(\{q_0, q_1, ...q_n\}, \{0,1\}, \delta, q_0, \{q_0\}\right)$

Since *M* recognizes language $C_n$. Hence, it is proved that $C_n$ is regular.

Comment

## Problem

An ***all***-NFA M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ if *every* possible state that M could be in after reading input x is a state from F. Note, in contrast, that an ordinary NFA accepts a string if *some* state among these possible states is an accept state. Prove that all-NFAs recognize the class of regular languages.

## Step-by-step solution

**Step 1** of 5

All NFA *M* is a 5-tuple $(Q,\Sigma,\delta,q_0,F)$ that accepts $x \in \Sigma^*$ if every possible state that M could be in after reading input *x* is a state from F.

There are two steps to prove that all – NFAs recognize the class of regular languages.

**Step 1:** Every regular language is recognized by some all – NFA

**Step 2:** Every all – NFA recognizes a regular language.

Comment

**Step 2** of 5

**Proof for the Step 1 as follows:**

Clearly every DFA can be viewed as an all – NFA.

• Let *L* be the any regular language.

• Let $M = (Q,\Sigma,\delta,q_0,F)$ be the DFA that recognizes *L*.

• Clearly for each input string *x*,

There is exactly one possible state $q \in Q$ that M could be in after reading *x*.

• Hence, if M is viewed as an all-NFA, then it accept *x* if and only if $q \in F$, which happens if and only if $x \in L$.

• Therefore, when M is viewed as an all – NFA, it also recognizes the language L.

Thus, every regular language is recognized by some all – NFA.

Comment

**Step 3** of 5

**Proof for the Step 2 as follows:**

• Let $N = (Q,\Sigma,\delta,q_0,F)$ be an all – NFA.

• Let *A* be the language recognized by N.

Now, prove that A is regular.

• Constructing a DFA $M = (Q',\Sigma,\delta',q_0',F)$ that recognizes *A*.

$M = (Q',\Sigma,\delta',q_0',F)$

• Where $Q' = P(Q)$, where $P(Q)$ is the set of subset of Q

• For $R \in Q'$ and $a \in \Sigma$

$$\delta'(R,a)=\bigcup_{r\in R}E(\delta(r,a))$$

• For any subset $S\subset Q, E(S)$ is the set of all sates $q\in Q$ that can reached from *S* by travelling along $\in$ arrows including the members of *S* themselves.

$$q_0'=E(\{q_0\})$$

$$F'=\{R\in Q' \mid R\subset F\}$$

Comment

**Step 4** of 5

Now, consider an input string $x\in\Sigma^*$.

• Let *R* be the set of states that the all $NFA$ *N* could be in after reading *x*.

• Then *x* is fed to the DFA *M* defined above, *M* will end at state *R*.

• By the definition of all-NFA , have $x\in A\Leftrightarrow R\subset F$.

• By the definition of *M*, *M* accepts *x* if and only if $R\in F'$, which is equivalent to $R\subset F$.

Hence M recognize A.

Comment

**Step 5** of 5

Therefore, every all- NFA recognizes a regular language.

From step 1 and step 2 all- NFAs recognizes the class of regular language.

Comment

## Problem

The construction inTheorem1.54 shows that every GNFA is equivalent to a GNFA with only two states. We can show that an opposite phenomenon occurs for DFAs. Prove that for every k > 1, a language $A_k \subseteq \{0,1\}^*$ exists that is recognized by a DFA with k states but not by one with only k – 1 states.

## Step-by-step solution

**Step 1** of 1

**Given:**

It is being given that if there is two states then GNFA is correspondingly equal to the GNFA. But, here user need to prove that in DFA subsequently opposite phenomenon occurs. It implies user need to prove that no DFA are equivalent to a DFA with lesser states.

**Proof:**

Assume $A_k$ be the set of words of length at least $k-1$. Therefore it can be said that $A_k$ has at least $k$ equivalence classes of words length $0,1,2,\ldots,k-2,\text{and } k-1$ or more. So it is clear from this that $A_k$ requires a DFA with $k$ states.

For any DFA fewer than $k$ states, by Pigeon Hole Principle, two of the $k$ strings cause the machine to loop in same state results in a rejection from the DFA.

Consider $A_k = \{0,1\}^* 0^{k-1} 0^*$ for $k > 1$.

Now a DFA with exactly $k$ states can recognize the language $A_k$. Starting from the start state there is a state in the DFA for each 0 it had read after the last 1.

After $k-1$ 0's it arrives at an accepting state whose further transitions are self-loops. Now based on the language let say $A_k$ be the set consisting of the strings $10,100,\ldots,10^{k-1}$. If the DFA consist of fewer than $k$ states then by the Pigeon Hole Principle two these strings cause a loop to a single state. Hence the machine fails to accept the strings.

**Conclusion:**

Hence, it can be said that no DFA's are equivalent to a DFA with lesser states.

Comment

## Problem

Recall that string x is a ***prefix*** of string y if a string z exists where xz = y, and that x is a ***proper prefix*** of y if in addition x ≠ y. In each of the following parts, we define an operation on a language A. Show that the class of regular languages is closed under that operation.

A **a.** $NOPREFIX(A) = \{w \in A \mid \text{no proper prefix of } w \text{ is a member of } A\}$.

**b.** $NOEXTEND(A) = \{w \in A \mid w \text{ is not the proper prefix of any string in } A\}$.

## Step-by-step solution

**Step 1** of 3

Consider the following information:

• String *x* is a prefix of string *y* if a string *z* exists such that *xz=y*.

• String *x* is a proper prefix of *y* if *xz=y* and $x \neq y$.

• The language *A* is regular language. Assume $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing *A*.

Comment

**Step 2** of 3

**a.**

$NOPREFIX(A) = \{w \in A \mid no\ proper\ prefix\ of\ w\ is\ a\ member\ of\ A\}$

1. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing *A*.

2. Initially, find all words that have a proper prefix in *A*. The language *L* is represented as $L = \{w \in \Sigma^* : x \in A\ and\ z \in \Sigma^*\ such\ that\ xz = y\}$.

3. Now, construct the NFA $M^1 = (Q^1, \Sigma, \delta^1, q_0^1, F^1)$ for all its components such that:

• $Q^1 = Q \cup \{q_f\}$ and $q_f \notin Q$

• For $q \in Q^1$ and $a \in \Sigma$ define $\delta^1(q,a) = \begin{cases} \delta(r,a) & if\ r \notin F \\ \phi & if\ r \in F \end{cases}$

• $q_0^1 = q_0$

• $F^1 = q_f$

**Proof:**

• If *w* is a string in Language *L*, there is a string *y* in *A*. Here, *x* is a proper prefix of *y* such that *xz=y* and *x* is non-empty.

• If *w* is taken as input of $M^1$, the computation on *x* ends at an accepting state in *M* and some computation on *z* ends at state $q_f$.

• So *w* is accepted by $M^1$, which means that there is a computation that ends at $q_f$.

• From the construction of $M^1$, the computation arrives at one of the accepting states in *M* before it reaches $q_f$.

• If we conclude that String *x* is a proper prefix of *y*, *M* on input *x* ends in one of its accepting states. So, *w* is a member of *L*, and *x* is in A.

• As, *NOPREFIX*(*A*) is defined as $A \cap \overline{L}$ and class of regular languages are closed under intersection and complement, *NOPREFIX*(*A*) is also regular.

Comments (2)

**b.**

$$NOEXTEND(A) = \{w \in A \mid w \text{ is not proper prefix of any string in } A\}$$

• Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing *A*.

• Assume that the DFA for language M accepts that only the strings reaching the final state but not those strings that are added to reach a final state again.

• So, the strings exactly ending in final states are accepted.

• For a state $q \in F$, check whether there is a path from $q \in Q$ to any state in *F* (or a cycle involving *q*) using Depth First Search.

• Let $F^{\text{I}} \subseteq F$ be the set of all the states from which there is no such path.

• Now, changing the set of final states *F* to $F^{\text{I}}$ gives a DFA for *NOEXTEND* (*A*).

• Thus, *NOEXTEND(A)* is also regular.

Comment

## Problem

For languages A and B, let the perfect shuffle of A and B be the language

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma\}.$$

Show that the class of regular languages is closed under perfect shuffle.

## Step-by-step solution

**Step 1** of 3

Consider the two languages *A* and *B*. The language *perfect shuffle* on *A* and *B* is as follows:
$\{w \mid w = a_1 b_1 ... a_k b_k, \text{ where } a_1 ... a_k \in A \text{ and } b_1 ... b_k \in B, \text{ each } a_i, b_i \in \Sigma\}$.

Assume, $DFA_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $DFA_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$ be two DFAs that recognize *A* and *B* respectively.
$DFA_{Perfect-shuffle} = (Q, \Sigma, \delta, S, F)$ recognizes the language perfect shuffle on *A* and *B*.

Comment

**Step 2** of 3

The DFA for perfect shuffle switches from $DFA_A$ to $DFA_B$ after each character is read and it tracks the current states of $DFA_A$ and $DFA_B$. Each character should belong to $DFA_A$ or $DFA_B$ i.e., $a_i, b_i \in \Sigma$. For each character read, $DFA_{Perfect-shuffle}$ makes moves in the corresponding DFA (either $DFA_A$ or $DFA_B$). After the whole string is read, if both $DFA_A$ and $DFA_B$ reaches to the final state, then the input string is accepted by $DFA_{Perfect-shuffle}$.

Comment

**Step 3** of 3

The $DFA_{Perfect-shuffle}$ is defined as follows:

- $Q = Q_A \times Q_B \times \{A, B\}$: set of all possible states of $DFA_A$ and $DFA_B$ which should match with $DFA_{Perfect-shuffle}$.
- The input alphabet for $DFA_{Perfect-shuffle}$ is $\Sigma$.
- $q = (q_A, q_B, A)$: $q_A$ and $q_B$ are the initial states for $DFA_A$ and $DFA_B$ respectively. $DFA_{Perfect-shuffle}$ starts with $q_A$ in $DFA_A$, $q_B$ in $DFA_B$ and the next character should be read from $DFA_A$.
- $F = F_A \times F_B \times \{A\}$: $F_A$ and $F_B$ are the final states for $DFA_A$ and $DFA_B$ respectively. $DFA_{Perfect-shuffle}$ accepts if both $DFA_A$ and $DFA_B$ reaches to the final states and the next character should be read from $DFA_A$.
- The transition function $\delta$ is,

1. $\delta((m, n, A), a) = (\delta_A(m, a), n, B)$
2. $\delta((m, n, B), b) = (m, \delta_B(n, b), A)$

Consider, the current state of $DFA_A$ is *m* and the current state of $DFA_B$ is *n*. Change the current state of *A* to $\delta_A(m, a)$ if the next character is to be read from $DFA_A$ when *a* is the next character. After the character is read, read the next character from $DFA_B$. Change the current state of *B* to $\delta_B(n, b)$ if the next character is to be read from $DFA_B$ when *b* is the next character.

The language *L* is said to be regular if there exist an FA that recognizes the language *L*. Here, the $DFA_{Perfect-shuffle}$ is defined for the language *perfect shuffle*.

**Therefore, the class of regular languages is closed under perfect shuffle.**

Comments (1)

## Problem

For languages A and B, let the ***shuffle*** of A and B be the language

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma^*\}.$$

Show that the class of regular languages is closed under shuffle.

## Step-by-step solution

**Step 1** of 2

Consider the two languages *A* and *B*. The language *shuffle* on *A* and *B* is as follows:
$\{w \mid w = a_1 b_1 ... a_k b_k, \text{ where } a_1 ... a_k \in A \text{ and } b_1 ... b_k \in B, \text{ each } a_i, b_i \in \Sigma^*\}$.

Assume, $DFA_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $DFA_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$ be two DFAs that recognize *A* and *B* respectively. $DFA_{shuffle} = (Q, \Sigma, \delta, S, F)$ recognizes the language perfect shuffle on *A* and *B*. For each character read, $DFA_{shuffle}$ may move from running $DFA_A$ to running $DFA_B$. The *NFA* is more flexible when compared to the *DFA*. In this case, $NFA_{shuffle} = (Q, \Sigma, \delta, S, F)$ has to be constructed to allow more flexibility.

The $NFA_{shuffle}$ keeps track the current states of $DFA_A$ and $DFA_B$. For each character read, $NFA_{shuffle}$ makes moves in the corresponding DFA (either $DFA_A$ or $DFA_B$). After the whole string is read, if both $DFA_A$ and $DFA_B$ reaches to the final state, then the input string is accepted by $NFA_{shuffle}$.

Comment

**Step 2** of 2

The $NFA_{shuffle}$ can be defined as follows:

- $Q = (Q_A \times Q_B) \cup \{q_0\}$ : The set of all possible states of $DFA_A$ and $DFA_B$ which should match with $NFA_{shuffle}$. Here, $q_0$ denotes the initial state.
- $q = q_0$
- $F = (F_A \times F_B) \cup \{q_0\}$: $F_A$ and $F_B$ are the final states for $DFA_A$ and $DFA_B$ respectively. The $NFA_{shuffle}$ accepts the string if both $DFA_A$ and $DFA_B$ are in accept states or $NFA_{shuffle}$ accepts the empty string.
- $\delta$ is as follows:

o $\delta(q_0, \varepsilon) = (q_A, q_B)$: At the start state $q_0$, the current state of $DFA_A$ is $q_A$ and the current state of $DFA_B$ is $q_B$ without reading anything.

o $(\delta_A(m,a), n) \in \delta((m,n), a)$: Change the current state of *A* to $\delta_A(m,a)$ when the character *a* is read. Here, the current state of $D_A$ is *m* and the current state of $D_B$ is *n*.

o $(m, \delta_B(n,a)) \in \delta((m,n), a)$: Change the current state of *B* to $\delta_B(n,a)$ when the character *a* is read. Here, the current state of $D_A$ is *m* and the current state of $D_B$ is *n*.

The language *L* is said to be regular if there exist an FA that recognizes the language *L*. Here, the $NFA_{shuffle}$ is defined for the language *shuffle*.

**Therefore, the class of regular languages is closed under shuffle.**

Comment

## Problem

Let A be any language. Define *DROP-OUT*(A) to be the language containing all strings that can be obtained by removing one symbol from a string in A. Thus,

$$DROP\text{-}OUT(A) = \{xz \mid xyz \in A \text{ where } x, z \in \Sigma^*, y \in \Sigma\}.$$

Show that the class of regular languages is closed under the *DROP-OUT* operation. Give both a proof by picture and a more formal proof by construction as in Theorem 1.47.

THEOREM **1.47**

The class of regular languages is closed under the concatenation operation.

## Step-by-step solution

**Step 1** of 4

Given that

*A* is any language and

$DROP_OUT(A) = \{xz \mid xyz \in A \text{ where } x, z \in \Sigma^*, y \in \Sigma\}$

We have to prove that class of regular languages closed under $\text{DROP_OUT}$ operation.

i.e. if *A* is a regular language then $DROP_OUT(A)$ is also regular.

We have to take that *A* is regular and we have to prove that $DROP_OUT(A)$ is regular.

Since *A* is a regular language, it must be recognized by a DFA.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA recognizes *A*.

Now we will construct an NFA $N = (Q', \Sigma \cup \{\in\}, \delta', q_0', F')$ that we recognize $DROP_OUT(A)$.

There are two copies of Machine *M*.

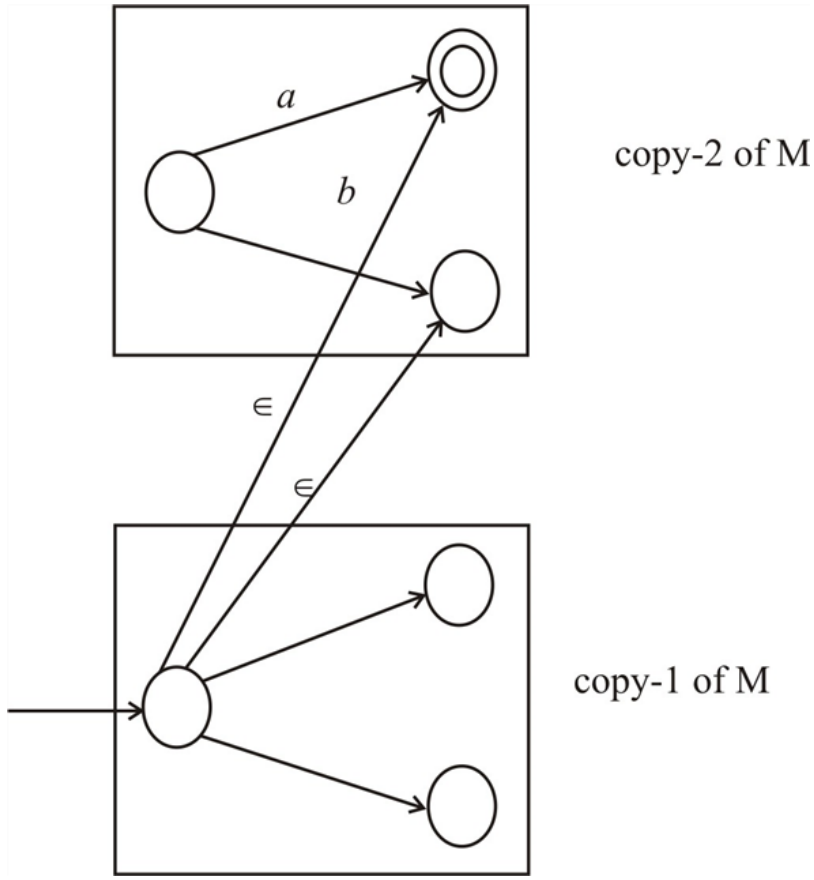
• Copy 1: Copy 1 corresponds to the state of having 'not yet skipped a symbol'

• Copy 2: Copy 2 corresponds to the state of having "already skipped a symbol".

Comment

**Step 2** of 4

(i) Proof by picture:-




Comment

**Step 3** of 4

$N=(Q',\Sigma\cup\{\in\},\delta',q_0',F')$

- $Q'=\{(q,b)\mid q\in Q,b\in\{0,1\}\}$ = set of states
- $q_0'=$ start state

$=(q_0,0)$

- $F'=$ set of final states

$=\{(q,1)\mid q\in F\}$

- $\delta'$ is gives as follows:

$\rightarrow\ \delta'((q,b),a)=\{(\delta(q,a),b)\}\ \forall q\in Q,b\in\{0,1\},a\in\Sigma$

This means that both the copy1 and copy2 of the machine *M* do exactly as the original machine does on every symbol a of the alphabet

$\rightarrow\ \delta'((q,0),\in)=\{(\hat{q},1)\mid\exists a\in\Sigma,\delta(q,a)=\hat{q}\}$

Also at every stage, the machine has the option to skip a character. The only accepting sates are in copy 2. This means, the machine cannot accept a string without skipping a character.

Comment

**Step 4** of 4

(ii) <u>Formal proof:</u>

$\rightarrow$ The formal proof is given by induction on the length the string.

$\rightarrow$ An appropriate inductive hypothesis is to assume that, for any string *w* of length *k*,

• The machine *M* stays in the copy -1 iff it has not yet skipped a symbol.

i.e. $\delta'^*((q_0,0),w)=(q_1,0)$ iff $\delta^*(q_0,w)=q_1$

• The machine *M* jumps to the copy-2 iff there is some symbol a that is skipped.

i.e. $\delta'^*((q_0,0),w)=(q_1,1)$ iff $\delta^*(q_0,w_1 a\, w_2)=q_1$.

So in both (i) and (ii) we constructed an NFA N that recognizes the language $DROP_OUT(A)$.

Thus $DROP_OUT(A)$ is regular.

Hence class of regular languages closed under $\mathrm{DROP_OUT}$ operation.

Comment

## Problem

Let B and C be languages over $\Sigma = \{0, 1\}$. Define

$$B \xleftarrow{1} C = \{w \in B \mid \text{ for some } y \in C, \text{ strings } w \text{ and } y \text{ contain equal numbers of 1s}\}.$$

Show that the class of regular languages is closed under the $\xleftarrow{1}$ operation.

## Step-by-step solution

**Step 1** of 2

Given that *B* and *C* are two languages and $B \xleftarrow{1} C = \{w \in B \mid \text{ for some } y \in C, \text{srings } w \text{ and } y \text{ contain equal numbers of } 1\}$ over the alphabet $\Sigma = \{0,1\}$

We have to prove that class of regular languages closed under $\xleftarrow{1}$ operation

That means if *B* and *C* are regular languages than $B \xleftarrow{1} C$ is also a regular language.

So given that *B* and *C* are regular languages.

We know that

"A language is regular if it is recognized by an automation"

• Let $M_B$ be the DFA that recognizes the language *B*

$M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$

• Let $M_c$ be the DFA that recognizes the language *C* $M_C = (Q_C, \Sigma, \delta_C, q_C, F_C)$

Now we have to construct an NFA which recognizes $B \xleftarrow{1} C$.

Comment

**Step 2** of 2

<u>Construction of NFA to recognize $B \xleftarrow{1} C$ :</u>

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA.

Now *N* has to decide whether a string $w \in B \xleftarrow{1} C$ or not.

• For that first machine M checks whether $w \in B$ or not.

• If $w \in B$, then non deterministically find out a string of that contains the same number of 1s as contained in *w* and checks that $y \in C$.

• That means for each string *B*, there are $C(\text{number ofstrings in } C)$ parallel machings will exist

Thus $Q =$ set of states

$= Q_B \times Q_C$

$\Sigma =$ set of alphabet

= same as B and C

$\delta$ is given by, for $(q, r) \in Q$ and $a \in \Sigma$

$$\delta(q,r),a=\begin{cases}\{(\delta_B(q,0),r)\} & \text{if } a=0\\ \{((\delta_B(q,1)),\delta_C(r,1))\} & \text{if } a=1\\ \{(q,\delta_C(r,0))\} & \text{if } a=\in\end{cases}$$

$q_0 =$ start state

$=(q_B,q_c)$

$F =$ set of final states

$= F_B \times F_C$

Thus we defined an NFA *N* to recognize $B \xleftarrow{1} C$.

Hence $B \xleftarrow{1} C$ is regular.

Therefore class of regular languages closed under $B \xleftarrow{1} C$ operation

Comments (3)

## Problem

Let $A/B = \{w \mid wx \in A \text{ for some } x \in B\}$. Show that if A is regular and B is any language, then A/B is regular.

## Step-by-step solution

**Step 1** of 2

Given language is

$A/B = \{w \mid wx \in A \text{ for some } x \in B\}$

Here, *A* is a regular language and *B* is any language.

Now, the objective is to prove that $A/B$ is regular.

Since *A* is a regular language, some DFA will recognize the language *A*.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA that recognizes *A*.

Here, *Q* is the set of states.

$\Sigma$ is set of alphabets = of the alphabets for *A* and *B*.

$\delta$ is the transition function.

$q_0$ is the start state.

$F$ is the set of final states.

Comment

**Step 2** of 2

To prove $A/B$ is a regular language, construct a DFA that recognizes the language $A/B$

Let $M' = (Q', \Sigma', \delta_0', F')$ be the DFA that recognizes $A/B$.

- $Q'$ = set of states = *Q*
- $\Sigma' =$ set of alphabets $= \Sigma$
- $\delta' =$ transition function $= \delta$
- $q_0' =$ start state $= q_0$
- $F' = \{q \in Q \mid \exists x \in B \text{ such that } M \text{ goes from } q \text{ to some sate in } F \text{ on reading } x\}$

Thus, a DFA $M'$ to recognize the language $A/B$ has been constructed.

Hence $A/B$ is a regular language.

Comment

## Problem

Prove that the following languages are not regular. You may use the pumping lemma and the closure of the class of regular languages under union, intersection, and complement.

**a.** $\{0^n 1^m 0^n \mid m, n \geq 0\}$

A **b.** $\{0^m 1^n \mid m \neq n\}$

**c.** $\{w \mid w \in \{0,1\}^* \text{ is not a palindrome}\}^8$

* **d.** $\{wtw \mid w, t \in \{0,1\}^+\}$

## Step-by-step solution

**Step 1** of 5

**Pumping lemma:**

For a regular language *A* with the pumping length *P*, if S is any string of *A* with minimum length of *P*, then the string S can be divided into 3 pieces *x, y* and *z* represented as $S = xyz$ should satisfy the following conditions:

- for each
- $|y| > 0$, and
- $|xy| \leq P$

Comment

**Step 2** of 5

**a.**

**Consider the Language** $L = \{0^n 1^m 0^n \mid m, n \geq 0\}$.

Assume that *L* is regular language and a string $S = 0^P 1 0^P$. Divide the string into three pieces *x, y* and *z*. So, $S = 0^P 1 0^P = xyz$ where, *P* is the pumping length.

Assume that $x = 0^{P-K}, y = 0^K$ and $z = 10^P$(where $K > 0$)

Now $xy^0 z = 0^{P-K}\left(0^K\right)^0 1\, 0^P$

$= 0^{P-K} 1 0^P \notin L \quad \left[\because y^0 = \in\right]$

The String $xy^0 z$ does not belong to *L* because *P-K < P.*

So, the assumption that *L* is regular is a contradiction. Thus, by using pumping lemma it is proved that *L* is not regular.

Comments (3)

**Step 3** of 5

**b**.

**Consider the Language** $L = \{0^m 1^n \mid m \neq n\}$ **.**

Assume that *L* is regular language and string $S = 0^P 1^{P+P!}$. Divide the string into three pieces *x, y* and *z*. So, $S = 0^P 1^{P+P!} = xyz \in L$ and $|S| \geq P$ where, *P* is the pumping length.

Note: *P!* is divisible by all integers from 1 to *P*, where $P! = P \times (P-1) \times (P-2) \times \ldots \times 1$.

Assume that $x = 0^a, y = 0^b$ and $z = 0^c 1^{P+P!}$, where $b \geq 1$ and $a + b + c = P$.

Now take string $s' = xy^{i+1}z$, where $i = \frac{P!}{b}$.

Then $y^i = 0^{P!}$ so $y^{i+1} = 0^{b+P!}$, and

So $xyz, = 0^{a+b+c+P!} 1^{P+P!}$.

That gives $xyz = 0^{P+P!} 1^{P+P!} \notin L$, a contridiction.

Here, $m = P + P!$, $n = P + P!$ and $m = n$. This is a contradiction to the assumption because $m = n$. Thus, by using pumping lemma it is proved *L* is not regular.

Comment

**Step 4** of 5

**c.**

**Consider the Language** $L = \{w \mid w \in \{0,1\}^* \text{ is not a palindrome}\}$ **.**

Assume that *L* is regular language.

The compliment of the language L is $\overline{L} = \{w \mid w \in \{0,1\}^* \textit{ is palindrome}\}$ is also regular.

Assume a string $S = 0^P 1\, 0^P$. Divide the string into three pieces *x, y* and *z*. So, $S = 0^P 1\, 0^P = xyz \in L$ where, *P* is the pumping length.

Assume that $x = 0^{P-K}, y = 0^K$ and $z = 10^P$ where $K > 0$

Now $xy^0 z = 0^{P-K} \left(0^K\right)^0 10^P$

$= 0^{P-K} 10^P \notin L \quad \left[\because y^0 = \in\right]$

The String $xy^0 z$ is not same from forward and backward direction because *P-K* < *P*.

So, the string $xy^0 z$ does not belong to $\overline{L}$. So, the assumption is a contradiction.

Thus, by using pumping lemma it is proved *L* is not regular.

Comment

**Step 5** of 5

**d.**

**Consider the Language** $L = \{wtw \mid w, t \in \{0,1\}^+\}$ **.**

Assume that *L* is regular language.

Assume a string $S = 0^P 1\, 0^P$. Divide the string into three pieces *x, y* and *z*. So, $S = 0^P 1\, 0^P = xyz \in L$ where, *P* is the pumping length.

Assume that $x = 0^{P-K}, y = 0^K$ and $z = 10^P$ where $K > 0$

Now $xy^0 z = 0^{P-K} \left(0^K\right)^0 10^P$

$= 0^{P-K} 10^P \notin L \quad \left[\because y^0 = \in\right]$

The String $xy^0 z$ does not belong to *L* because *P-K* < *P* and not of the form *wtw* as in *L*.

So, the assumption is a contradiction. Thus, by using pumping lemma it is proved *L* is not regular.

Comments (3)

## Problem

Let $\Sigma = \{1, \#\}$ and let

$$Y = \{w \mid w = x_1\#x_2\#\cdots\#x_k \text{ for } k \geq 0, \text{ each } x_i \in 1^*, \text{ and } x_i \neq x_j \text{ for } i \neq j\}.$$

Prove that Y is not regular.

## Step-by-step solution

**Step 1** of 4

Consider the following language over the alphabet $\Sigma = \{1, \#\}$.

$Y = \{w \mid w = x_1 \# x_2 \# ... \# x_k \text{ for } k \geq 0, \text{ each } x_i \in 1^*, \text{ and } x_i \neq x_j \text{ for } i \neq j\}$. The language *Y* accepts the words of the form $x_1 \# x_2 \# ... \# x_k$ where $x_1, x_2, \ldots x_k$ are the substrings that are formed with any number of 1s. Here, $x_i$ can be any number of 1s.

The words that are accepted by the language *Y* contains only 1s and #s because $\{1, \#\}$ are the input alphabet. Any two substrings cannot be equal i.e., $x_i \neq x_j$. Every two substrings are separated by the input alphabet #.

The language is said to be regular if it is satisfied by the pumping lemma. Otherwise the language is not regular.

Comment

**Step 2** of 4

**Pumping lemma:**

If *A* is a regular language, then there is a number *p* (the pumping length) where *S* is any string that belongs to *A* of length at least *p*, then *S* may be divided into three pieces, $S = uvw$, satisfying the following conditions.

1. For each $i \geq 0, uv^i w \in A$
2. $|v| > 0$, and
3. $|uv| \leq p$

Comment

**Step 3** of 4

Assume that $Y$ is a regular language.

Let $p$ be the pumping length for $Y$. The strings of the language $Y$ are of the form $w = x_1 \# x_2 \# ... \# x_k$.

Consider a string $S = x_1 \# x_2$ for $k = 2$ and $x_1 \neq x_2$. Here, $x_1$ and $x_2$ can be formed with only 1s but both cannot be equal. Any two different strings can be taken for $x_1$ and $x_2$.

Assume $x_1 = 1^p 1$ and $x_2 = 1 1 1^p$. Then the string $S = 1^p 1 \# 1 1 1^p$. Here, $x_1$ and $x_2$ are two different substrings and the value of $x_1$ and $x_2$ depends on the *p* value. For example, if *p*=2 then the values of $x_1$ and $x_2$ are 111 and 1111.

Clearly, the length of *S* is greater than *p* and $S \in Y$.

Comment

**Step 4** of 4

Let $111\#1111$ be the string that belongs to $Y$. The pumping length of the string is 2.

To satisfy the conditions of the pumping lemma, divide the string $111\#1111$ into three parts *u*, *v* and *w*. Here *u* is equal to 1, *v* is equal to 1, *w* is equal to $1\#1111$ (the remaining part of the string).

$$\begin{aligned} S &= 111\#1111 \\ &= \underset{u}{\underline{1}}\ \underset{v}{\underline{1}}\ \underset{w}{\underline{1\#1111}} \end{aligned}$$

Pump the middle part such that $uv^iw\ (i \geq 0)$. For *i*=2, the *v* becomes 11.

$$\begin{aligned} S &= (1)\ (1)^i\ (1\#1111) \\ &= \underset{u}{\underline{1}}\ \underset{v}{\underline{11}}\ \underset{w}{\underline{1\#1111}} \qquad [when\ i{=}2] \end{aligned}$$

The string after pumping is $1111\#1111$.

The string $1111\#1111 \notin Y$ because, the substring $x_1$ is equal to $x_2$ which violates the condition of the language *Y*. It is a contradiction. Thus, the pumping lemma is violated.

**Therefore, $Y$ is not a regular language.**

Comment

## Problem

Let ∑ = {0,1} and let

D = {w|w contains an equal number of occurrences of the substrings 01 and 10}

Thus 101 $\in$ D because 1010 contains two 10s and one 01. Show that D is a regular language.

## Step-by-step solution

**Step 1** of 2

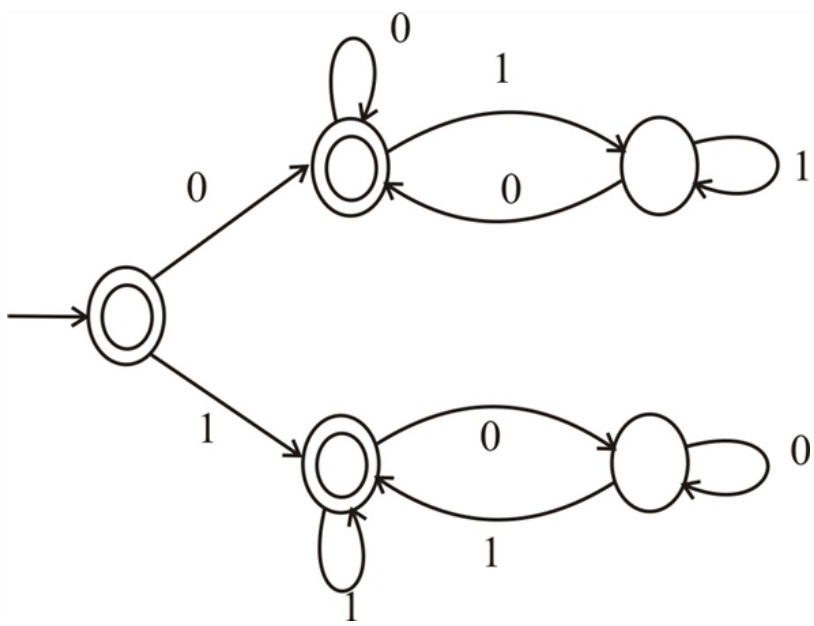
Given language is

$D = \{w \mid w \text{ contains an equal number of occurrences of the substring } 01 \text{ and } 10\}$ over the alphabet $\Sigma = \{0,1\}$

We have to prove that *D* is a regular language.

A language is regular if some DFA recognizes it.

The following DFA recognizes the language *D*.




Comment

**Step 2** of 2

This DFA recognize the language *D*.

Thus *D* is regular

Comment

## Problem

**a.** Let

$$B = \{1^k y \mid y \in \{0,1\}^* \text{ and } y \text{ contains at least } k \text{ 1s, for } k \geq 1\}.$$

Show that B is a regular language.

**b.** Let

$$C = \{1^k y \mid y \in \{0,1\}^* \text{ and } y \text{ contains at most } k \text{ 1s, for } k \geq 1\}.$$

Show that C isn't a regular language.

## Step-by-step solution

**Step 1** of 2

**(a)**

**The language given is as follows:**

$B = \{1^k y \mid y \in \{0,1\}^*\}$ and *y* contains at least k 1s, for $k \geq 1\}$

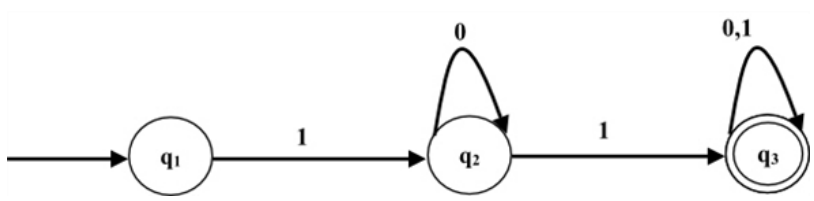
From the definition 1.16: A language is said to be Regular language if some finite automaton recognizes it.

String in language B must start with a 1 and contains at least one other 1, if k=1. So, if k is positive that, any string that start with a 1 and contains at least one other 1 matches in the y. B is defined by regular expression $10^*1(1 \cup 0)^*$ and therefore B is regular.

To show *B* is regular, the **definition 1.16** can be used.

Let *M* be the DFA recognizes the language *B*.

The state diagram of *M* is as follows:




Since there is a DFA recognizing the language *B*.

**Hence it is proved that *B* is regular language.**

Comments (4)

**Step 2** of 2

**(b)**

**The language given is as follows:**

$C = \{1^k y \mid y \in \{0,1\}^* \text{ and } y \text{ contains at most } k \text{ 1s, for } k \geq 1\}$

The proof for to prove *C* is not regular language is as follows:

Assume that *C* is a regular language.

Assume *P* as the pumping length.

Consider a string $S = 1^P 0 1^P \in C$

Using the pumping lemma, *S* can be written as

$S = 1^P 0 1^P = uvw$ such that $|uv| \le P, |y| > 0$ and $uv^i w \in C \;\; \forall i \ge 0$

When $i = 0, uw = 1^{P-t} 0 \;\; 1^P$ for some *k* where the number of 1's in *y* is less than or equal to *k*. $1^k y = 1^{P-t} 0 1^P$.

From this, *y* must contain the substring $0 1^P$.

So $k \le P - t$ with the number of 1s in $y \ge P$.

So, the number of 1's in *y* is always greater than *k* since $t > 0$.

Therefore, it is proved that $uw$ does not belong to language *C*.

Since the above statement results a contradiction.

**Hence, it is proved that the given language *C* is not a regular language.**

Comment

## Problem

Read the informal definition of the finite state transducer given in Exercise 1.24. Prove that no FST can output $w^R$ for every input w if the input and output alphabets are {0,1}.

## Step-by-step solution

**Step 1** of 1

Let $w = w_1 w_2 \ldots w_n$ then reverse of $w$ can be written as $w^R = w_n \ldots w_2 w_1$

We have to prove that no FST(Finite state transducer) can output $w^R$ for every input $w_1$ over the alphabets $\{0,1\}$

Let us assume that FST *T* output $w^R$ on input *w*.

Consider two input strings 00 and 01

(i) On input 00, FST *T* must produce output 00 because T outputs $w^R$ on input *w*.

(ii) On input 01, FST *T* must produce output 10 because T outputs $w^R$ on input *w*.

In these both case the first input bit is same i.e., 0.

But first output bits differ (0 and 1).

But from actual definition of FST, FST can produce its first output bit before it reads its second input.

Thus FST cannot operate to produce $w^R = 10$ on input $w = 01$.

Thus our assumption that FST *T* outputs $w^R$ on input *w* is wrong.

Hence no FST outputs $w^R$ on input *w*.

Comment

## Problem

Let x and y be strings and let L be any language. We say that x and y are ***distinguishable by*** L if some string z exists whereby exactly one of the strings xz and yz is a member of L; otherwise, for every string z, we have xz $\in$ is an equivalence relation.

## Step-by-step solution

**Step 1** of 1

*L* be any language and $x$ and $y$ are the strings

***Distinguishable by*** $L$ :

Strings $x$ and $y$ are distinguishable by $L$ if $\exists$ some $z$ such that exactly one of the strings $xz$ and $yz$ belongs to $L$.

***Indistinguishable by*** $L$ :

Strings $x$ and $y$ are indistinguishable by $L$ if for every string $z$, $xz \in L$ whenever $yz \in L$

If $x$ and $y$ are indistinguishable by $L$ then we write $x \equiv_L y$

Now we have to show that $\equiv_L$ is an equivalence relation.

• To show that $\equiv_L$ is an equivalence relation, we have to show that $\equiv_L$ is

(i) Reflexive

(ii) Symmetric

(iii) Transitive

• According to the given data, $x \equiv_L y$ means "for every string *z*, *xz* is in $L$ whenever *yz* is in $L$". That means, "for every string $z, xz$ is in $L$ iff $yz$ is in $L$"

(i) Reflexivity: $x \equiv_L x$ is true

For all strings $z, xz$ is in $L$ iff $xz$ is in $L$

Therefore $x \equiv_L x$ is true.

Hence $\equiv_L$ is reflexive.

(ii) Symmetry: $x \equiv_L y$ implies $y \equiv_L x$

If $x \equiv_L y$ is true then "for all *z*, *xz* is in $L$ iff *yz* is in $L$"

Which is equivalent to "for all *z*, *yz* is in $L$ iff *xz* is in $L$"

Therefore $y \equiv_L x$ is also true.

Hence $\equiv_L$ is symmetric.

(iii) Transitivity: If $a \equiv_L b$ and $b \equiv_L c$ then $a \equiv_L c$

This means that

"for all $z, az$ is in $L$ iff $bz$ is in $L$ and

For all $z, bz$ is in $L$ iff $cz$ is in $L$".

Therefore, "for all $z, az$ is in $L$ iff $cz$ is in $L$".

That is, $a \equiv_L c$ is true

Hence $\equiv_L$ is transitive.

From (i), (ii) and (iii)

$\equiv_L$ is equivalence relation.

Comment

## Problem

**Myhill–Nerode theorem.** Refer to Problem 1.51. Let L be a language and let X be a set of strings. Say that X is ***pairwise distinguishable by*** L if every two distinct strings in X are distinguishable by L. Define the ***index of*** L to be the maximum number of elements in any set that is pairwise distinguishable by L. The index of L may be finite or infinite.

**a.** Show that if L is recognized by a DFA with k states, L has index at most k.

**b.** Show that if the index of L is a finite number k, it is recognized by a DFA with k states.

**c.** Conclude that L is regular iff it has finite index. Moreover, its index is the size of the smallest DFA recognizing it.

## Step-by-step solution

**Step 1** of 4

The definition of **Myhill-Nerode theorem** is as follows:

**Myhill–Nerode theorem:** for any language $L$

- ***Distinguishable by L:*** *x* and *y* are the strings ***distinguishable*** by $L$, for the string *z* in generating of the strings $xz$ or $yz$ is a member of $L$.
- ***Indistinguishable by L:*** *x* and *y* are ***indistinguishable*** by $L$ for the string *z* we have $xz \in L$ every time $yz \in L$. We can write $x \equiv_L y$.
- ***Pair-wise distinguishable by L:*** set of strings contains in $S$**,** if every two separate strings are distinguishable in $L$.
- ***Index of L:*** It can count as finite or infinite. Language $L$ contains max number of elements which are ***pair-wise distinguishable***.

Comment

**Step 2** of 4

(a) Language $L$ recognized by DFA (Deterministic Finite Automata) as $M$ with number of states is $k$. We have to prove that $L$ has an index at most $k$.

Take a contradiction assumption i.e., $L$ has an index greater than $k$.

If $L$ contains index more than $k$ then $k+1$ strings are at least in any set $S$ which is ***pair wise distinguishable by*** $L$.

**Pigeonhole's** principle:

We will find two distinct strings $x$ and $y$ from $S$, such that the state of DFA $M$ after reading input $x$ is the same as the state of DFA $M$ after reading input $y$.

By applying **Pigeonhole's** principle both $xz$ and $yz$ are not in $L$. This is not satisfying the definition ***Distinguishable by L*** in **Myhill-Nerode theorem**

Hence contradiction occurs. Therefore our assumption that $L$ has index greater than $k$ is wrong. So, $L$ has index at most $k$.

Comment

**Step 3** of 4

(b) Index of Language $L$ contains $k$ finite states i.e., set $S = \{s_1, s_2, ...s_k\}$. We have to prove that $L$ recognized by DFA with $k$ states.

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be *DFA* with *k* states that recognizes $L$
- The construction of $M$ is as follows:

o Assume $Q = (q_1, q_2 ... q_k)$ is the set of states.

o Transition function is given as: $\delta(q_i, a) = q_j$ if $s_i a$ and $s_j$ are not distinguishable.

o $F = \{q_i \mid s_i \in L\}$ be the setoff

o Start state $q_0$ be the state such that $s_i$ and the empty string $\in$ are not distinguishable by $L$.

• We show that if string *t* and $s_j$ are not distinguishable by $L$, the state of $M$ will be $q_j$ after reading *t* as input.

• By the definition of *F*, $M$ accepts *t* if and only if *t* is in $L$.

• Hence $M$ recognizes $L$.

Comment

**Step 4** of 4

(c) Language $L$ is regular if it contains finite index. Index is size of smallest DFA recognizing it.

(i) <u>if $L$ is regular then $L$ has finite index:</u>

• Let us assume that $L$ is regular.

• $M$ be *DFA* that recognizes $L$.

• Let $k$ be the number of states in $M$.

• Then from part (a), $L$ has index at most $k$

(ii) <u>if $L$ has finite index then $L$ is regular:</u>

• Let us assume that $L$ has finite index $k$

• Then from part (b) we can contract a *DFA* with *k* states recognizing $L$

• We know that "A language is regular if and only if it is recognized by some *DFA*"

• Therefore $L$ is regular language.

Therefore from (i) and (ii) *L* is regular if and only if it has finite index.

• The index $k$ is size of the smallest *DFA* $M$ recognizing it, suppose on the opposing that is not true. From part (a) we could terminate that $L$ has indexed fewer than $k$, which contradicts fact that $L$ has index equal to $k$.

Comment

## Problem

Let ∑ = {0, 1, +, =} and

ADD = {x=y+zl x, y, z are binary integers, and x is the sum of y and z}.

Show that ADD is not regular.

## Step-by-step solution

**Step 1** of 3

Consider the following language over the alphabet $\Sigma = \{0,1,+,=\}$.

$ADD = \{x = y + z \mid x, y, z \text{ are binary integers, and } x \text{ is the sum of } y \text{ and } z\}$.

The language is said to be regular if it is satisfied by the pumping lemma, otherwise the language is not regular.

Comments (3)

**Step 2** of 3

**Pumping Lemma:**

If *A* is any regular language then there is a number $p$ ( the pumping length) where *S* is any string that belongs to *A* of length at least *P*, then *S* may be divided into three pieces, $S = uvw$, satisfying the following conditions.

1. For each $i \geq 0, uv^i w \in A$
2. $|v| > 0$, and
3. $|uv| \leq p$

Comment

**Step 3** of 3

Assume that $ADD$ is a regular language.

Let $p$ be the pumping length given by the pumping lemma. The strings of the language *ADD* are of the form $x = y + z$. Consider a string $1^{p+1} = 10^p + 1^p \in ADD$.

Let $111 = 100 + 11$ be the string that belongs to *ADD*. The pumping length of the string is 2. To satisfy the conditions of the pumping lemma, divide the considered string into three parts. Here, *u* is equal to 1, *v* is equal to 1, *w* is equal to 1=100+11 (the remaining part of the string).

$$\begin{aligned} S &= 111 = 100 + 11 \\ &= \underset{u}{1}\ \underset{v}{1}\ \underset{w}{1 = 100 + 11} \end{aligned}$$

Pump the middle part such that $uv^i w\ (i \geq 0)$. For *i*=2, the *v* becomes 11.

$$\begin{aligned} S &= (1)\ (1)^i\ (1 = 100 + 11) \\ &= \underset{u}{1}\ \underset{v}{11}\ \underset{w}{1 = 100 + 11} \qquad [when\ i=2] \end{aligned}$$

The string after pumping is $1111 = 100 + 11$.

The string $1111 = 100 + 11 \notin ADD$ because, addition of two binary numbers 100, 11 is not equal to the binary number 1111. It is a contradiction. So, the pumping lemma is violated.

**Therefore, $ADD$ is not a regular language.**

Comments (1)

## Problem

Consider the language $F = \{\mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}.$

**a.** Show that F is not regular.

**b.** Show that F acts like a regular language in the pumping lemma. In other words, give a pumping length p and demonstrate that F satisfies the three conditions of the pumping lemma for this value of p.

**c.** Explain why parts (a) and (b) do not contradict the pumping lemma.

## Step-by-step solution

**Step 1** of 4

**Consider the language**:

$F = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}$

The language F is the union of three disjoint languages.

$F_0 = \{b^j c^k \mid j, k \geq 0\}, F_1 \{ab^j c^j \mid j \geq 0\}, F_2 = \{a^i b^j c^k \mid i \geq j, k \geq 0\}$

Clearly $F_0$ and $F_2$ are regular languages.

The class of regular languages is closed under union and complement.

Thus $\overline{F_0 \cup F_2}$ is also regular.

We have $F_1 = F - (F_0 \cup F_2)$

$= F \cap \overline{F_0 \cup F_2}$

Since the class of regular languages is closed under intersection if *F* is regular, then so is $F_1$.

Comments (1)

**Step 2** of 4

**a.**

Use pumping lemma to show that $F_1$ is not regular and hence neither is F.

Assume that $F_1$ is regular language.

Let *P* be the pumping length given by pumping lemma.

Consider a string $S = ab^P c^P \in F_1$

Clearly $|S| > P$.

By using the pumping lemma, *S* can be divided into three pieces.

i.e., $S = ab^P c^P = uvw$ such that $|uv| \leq P, |v| > 0$ and $uv^i w \in F_1 \quad \forall i \geq 0$.

Take $u = a \quad v = b^P \quad w = c^P$

$uv^0 w = a(b^P)^0 c^P \quad \therefore (i = 0)$

$= ac^P \notin F_1$

The string *w* consist of '*c*'s. In this case, string $a\left(b^{P}\right)^{0}c^{P}$ has more *c*s than '*b*'s and so is not a member of $F_1$, violating the condition of pumping lemma.

This is contradiction. The previous assumption that $F_1$ is regular is wrong. Thus $F_1$ is not regular.

Therefore, *F* is also not a regular language.

Comments (1)

**Step 3** of 4

**b.**

Let pumping length $P = 2$

• Show that every string $S \in F$ of length at least *P* can be divided into three pieces $S = uvw$ such that, $|uv| \le P, |v| > 0$ and $uv^l w \in F \, \forall l \ge 0$.

• Consider a string $a^i b^j c^k \in F$ of length at least 2.

• Choose *x* to be the empty string.

• If $i \ne 2$, then choose *y* to be the first symbol in $a^i b^j c^k$.

• If $i = 2$, then choose $y = aa$

Clearly these chosen $x, y$ satisfies the three conditions of pumping lemma.

Comment

**Step 4** of 4

**c.**

part (a) and part(b) do not contradict the pumping lemma because the pumping lemma just says if a language is regular then there is a pumping length for that language. But, if the language satisfies the pumping lemma, the language may not be regular.

Comment

## Problem

The pumping lemma says that every regular language has a pumping length p, such that every string in the language can be pumped if it has length p or more. If p is a pumping length for language A, so is any length p' ≥ p. The ***minimum pumping length*** for A is the smallest p that is a pumping length for A. For example, if A = $01^*$, the minimum pumping length is 2. The reason is that the string s = 0 is in A and has length 1 yet s cannot be pumped; but any string in A of length 2 or more contains a 1 and hence can be pumped by dividing it so that x = 0, y = 1, and z is the rest. For each of the following languages, give the minimum pumping length and justify your answer.

A **a.** $0001^*$

A **b.** $0^*1^*$

**c.** $001 \cup 0^*1^*$

A **d.** $0^*1^+0^+1^* \cup 10^*1$

**e.** $(01)^*$

**f.** $\varepsilon$

**g.** $1^*01^*01^*$

**h.** $10(11^*0)^*0$

**i.** $1011$

**j.** $\Sigma^*$

## Step-by-step solution

**Step 1** of 10

**a.**

The pumping length could not be 3 as 000 being in the language it cannot be pumped. Consider the string length be 4 or more and divide in $xyz$ as $x$ being $000$, $y$ being the first $1$ and $z$ being everything after then it satisfies every condition of pumping lemma.

**Hence the minimum pumping length is 4.**

Comment

**Step 2** of 10

**b.**

String $\varepsilon$ is in the language but it could not be pumped so the pumping length could not be 0. According to pumping lemma's three condition arises which are as follows:

- If user divide the string in $xyz$ as $x$ is $\varepsilon$
- $y$ is first symbol $(0|1)$
- $z$ being the everything after then it holds.

**Hence the minimum pumping length is 1.**

Comment

**Step 3** of 10

**c.**

The string $001$ is in the language but if it is generated by **001** then it cannot be pumped. If string $s$ is larger than 3 and in the language, then it is generated by $0^*1^*$. Dividing the string according to Pumping Lemma's condition into a string $xyz$ where $x$ is $\varepsilon$, $y$ be the first symbol and $z$ be the remaining, user can pump the string.

**Hence the minimum pumping length is 4.**

Comment

**Step 4** of 10

**d.**

The string $11$ is in the language but it cannot be pumped so the length is not 2. Let $s$ be the string in the language of length at least 3. If $s$ is generated by $0^*1^+0^+1^*$ it can be divided in $xyz$ as $x$ is $\varepsilon$, $y$ is the first string and z is everything else so it can be pumped.

Again if $s$ is generated by $10^*1$ then also user can write it as $xyz$ where $x$ is $1$, $y$ is $0$, and $z$ is the remainder so it could be pumped.

**Hence the minimum pumping length is 3.**

Comment

**Step 5** of 10

**e.**

Let $s$ be a string in the language.

Now $s$ could be $\varepsilon$ but it cannot be pumped so the length is not 0. Next $s$ could be $01$ which if divide in $xyz$ as $x$ is empty string $\varepsilon$, $y$ is $01$, and $z$ is everything after then it satisfies the three conditions of pumping lemma.

Pumping length is not 1 because since there is no string of length 1 in the language.

**Hence the minimum pumping length is 2.**

Comment

**Step 6** of 10

**f.**

Let $s$ be a string in the language then $s$ is $\varepsilon$ and according to pumping lemma it cannot be pumped. As per the pumping lemma, pumping length should greater then equal to 1.

**Hence the minimum pumping length is 0.**

Comments (1)

**Step 7** of 10

**g.**

The minimum pumping length of the language could not be 2 as $00$ being in the language it could not be pumped. Let $s$ be the string of length at least 3 in the language so minimally $s$ could be $100$ or $010$ or $001$.

Now dividing $s$ in $xyz$ in all the three cases user get, for $100$ $x$ is $\varepsilon$, $y$ is $1$, and $z$ is the remainder, for $010$ $x$ is $0$, $y$ is $1$, and $z$ is the remainder and for $001$ $x$ is $00$, $y$ is $1$, and $z$ is the remainder. All satisfies Pumping Lemma's three conditions.

**Hence the minimum pumping length is 3.**

Comment

**Step 8** of 10

**h**.

The minimum length string in the given language is $100$ but it cannot be pumped. Next minimum length string is 10100. If divide it according to the pumping lemma in $xyz$ then $x$ be 10, $y$ be 10, and $z$ be the rest of the string then $y$ can be pumped.

**Hence the minimum pumping length is 4.**

Comment

**Step 9** of 10

i.

If $s$ be a string in the language then $s$ is $1011$. If set p = 4, then claim $s$ is pumpable (which it is not, as it is the only string in the language). This should be 5.

**Hence then the minimum pumping length is 5.**

Comment

**Step 10** of 10

**j.**

Say $s$ be a string in the language $\Sigma^*$. According to pumping lemma if divide $s$ in $xyz$ then $x$ be the empty string, $y$ is $(\varepsilon \mid 0 \mid 1)$ and $z$ is empty string. Now $\varepsilon$ could not be pumped.

**Hence the minimum pumping length is 1.**

Comment

## Problem

If A is a set of natural numbers and k is a natural number greater than 1, let

$B_k(A)$ = {wl w is the representation in base k of some number in A}.

Here, we do not allow leading 0s in the representation of a number. For example, $B_2(\{3, 5\}) = \{11, 101\}$ and $B_3(\{3, 5\}) = \{10, 12\}$. Give an example of a set A for which $B_2(A)$ is regular but $B_3(A)$ is not regular. Prove that your example works.

## Step-by-step solution

**Step 1** of 1

**Regular and Non Regular Expression**

Assume $\begin{aligned} A &= \{2n \mid \text{n is a natural number}\} \\ &= \{2,\ 4,\ 8,\ 16,\ 18,\ 20\ldots\} \end{aligned}$

So $B_2(A) = \{10,\ 100,\ 1000, 10000\ldots\}$ should be regular, because $B_2(A)$ is recognized by regular expression $10*$.

Now $B_3(A) = \{2,\ 11,\ 22,\ 121\ldots\}$

$B_3(A)$ is non regular.

It can be proved by contradiction that $B_3(A)$ is non regular.

Take on the contrary that $B_3(A)$ is regular.

Now $p$ will be pumping length according to pumping lemma.

Choose $u$ as element of $B_3(A)$ and the length of $u$ should at least $p+1$.

Since $u \in B_3(A)$ and $i > 1$, by pumping lemma $u$ can be divided into three part, $u = xyz$ where $\forall i \geq 0$ the string $xy^iz \in B_3(A)$

According to condition three of pumping lemma, $|xy| \leq p$, and $|z| > 0$.

If rightmost digit of $z$ is $0$, then $u$ will be the power of 3. But $u$ is power of 2, hence it is impossible. Hence the rightmost digit of $0$ will must be 1 or may be 2.

For $i > 1$, $u' = xy^iz$ is power of $2$ which is greater than $u$, so it is easy to generate it after $u$ is added to itself few numbers of times. That is, if $u$ is added at least 3 times to itself, there must be carry from the right to left column of $z$. When $i$ will increase, the carries will affect more columns. For very large $i$, the carries will bleed on $y$, as pumping lemma condition two says that $y$ should not be empty.

Power of 2 will be generated for very large value of $i$ which cannot be generated by copying $y$ to $i$ times. The carries will force $y$ as well as $z$ to change. Hence, it is showing that $B_3(A)$ does not satisfy pumping lemma. Hence, the initial assumption that $B_3(A)$ is regular, is wrong.

Hence, $B_3(A)$ is non-regular.

Comment

## Problem

If A is any language, let $A_{\frac{1}{2}-}$ be the set of all first halves of strings in A so that

$$A_{\frac{1}{2}-} = \{x \mid \text{ for some } y,\ |x| = |y| \text{ and } xy \in A\}.$$

Show that if A is regular, then so is $A_{\frac{1}{2}-}$.

## Step-by-step solution

**Step 1** of 2

Consider the language *A* is regular. Let $A_{\frac{1}{2}-}$ be the set of all first halves of strings in *A*.

$$A_{\frac{1}{2}-} = \{x \mid \text{ for some } y,\ |x| = |y| \text{ and } xy \in A\}$$

Since *A* is a regular language, the DFA *M* recognizes the language *A*.

$$M = (Q, \Sigma, \delta, q_0, F)$$

where, $Q$ is the set of states

$\Sigma$ is the input alphabet

$\delta$ is the transition function

$q_0$ is the start state

$F$ is the final state

The language is said to be regular if there exists an FA for it. In this case, construct an NFA *N* that recognizes $A_{\frac{1}{2}-}$. Let *x* be the first part and choose *y* such that $|x| = |y|$. Here, $x \in A_{\frac{1}{2}-}$. To prove the language $A_{\frac{1}{2}-}$ is regular, run two DFAs at the same time one forward and the other backward. Run the DFA *M* on input *x* in forward direction and run the DFA *M* on input *y* in backward direction parallelly. The input string is accepted if both simulations reach the same state.

Comments (1)

**Step 2** of 2

**Construction of NFA N to recognize $A_{\frac{1}{2}-}$ :**

Let $N = (Q', \Sigma, \delta', q_0', F')$ where,

(i) $Q' = Q \times Q \cup \{q_0'\}$ set of states contains the following:

• Special start state $q_0'$ and

• A cross product $q \times q \times q$ where

• The first part tracks the performance of *M* on *x*

• The second part does the same thing for *y*.

• The third part tests whether the guess on *M* is consistent or not.

(ii) $\Sigma =$ input alphabet

(iii) $q_0'$ is the start state

(iv) $F' =$ set of final states $= \left\{ \left\langle q_i, q_j, q_k \right\rangle \mid q_i, q_k, q_j \in Q \right\}$

(v) $\delta' =$ Rules of transition are as follows:

• There exists an $\varepsilon$ move from the start state to the all the states in $\{(q_0, q_f) \mid q_f \in F\}$.

• Consider the states $q_i, q_j, q_k, q_l \in Q$. There exists a move from $(q_i, q_j)$ to $\left(q_k, q_l\right)$ on input symbol $a \in \Sigma$ if and only if $\delta(q_i, a) = q_k$ and $\delta(q_l, b) = q_j$.

The NFA *N* is constructed to recognize $A_{\frac{1}{2}^-}$. Thus, $A_{\frac{1}{2}^-}$ is regular.

Therefore, if *A* is regular then $A_{\frac{1}{2}^-}$ is also regular.

Comment

## Problem

If A is any language, let $A_{\frac{1}{3}-\frac{1}{3}}$ be the set of all strings in A with their middle thirds removed so that

$$A_{\frac{1}{3}-\frac{1}{3}} = \{xz \mid \text{ for some } y,\ |x| = |y| = |z| \text{ and } xyz \in A\}.$$

Show that if A is regular, then $A_{\frac{1}{3}-\frac{1}{3}}$ is not necessarily regular.

## Step-by-step solution

**Step 1** of 2

**Constraints:**

$A$ is any language and $A_{\frac{1}{3}-\frac{1}{3}} = \{x \mid \text{ for some } y, |x| = |y| = |z| \text{ and } xyz \in A\}$ is the set of all strings in $A$ with their middle thirds removed.

• Let $A = \{a*\#b*\}$ is regular language

• We know that $\{a*b*\}$ is a regular language.

• Also we know that "Regular languages are closed under intersection"

• Now $A_{\frac{1}{3}-\frac{1}{3}} \cap \{a*b*\} = \{a^n b^n \mid n \geq 0\}$

• Clearly $\{a^n b^n \mid n \geq 0\}$ is not regular, because if $p$ is the pumping length and

$S = xyz = aabb$ is $p = 2$ Here $x = a$ $y = a$ $z = bb$, obtain $xy^2z = aaabb$

Comment

**Step 2** of 2

**Pumping lemma :** If $A$ is a regular language, then there is a pumping length $p$ where, if $s$ is any string in $A$ of length at least $P$, then $s$ may be divided into three pieces, $s = xyz$, satisfying following conditions

(i) For each $i \geq 0, xy^iz \in A$

(ii) $|y| > 0$ and

(iii) $|xy| \leq p$

So according to pumping lemma $xy^2z = a^3b^2 \notin \{a*b*\}$

Hence $\{0*1*\}$ is not regular.

As regular languages are closed under intersection and $\{0*1*\}$ is not regular, $A_{\frac{1}{3}-\frac{1}{3}}$ is not regular. If $A$ is regular, then $A_{\frac{1}{3}-\frac{1}{3}}$ is not necessarily regular is proved.

Comment

## Problem

Let $M = (Q, \Sigma, \delta, q_0, F)$ (q, s) equals the state where M ends up when M starts at state q and reads input s.) Say that M is ***synchronizable*** if it has a synchronizing sequence for some state h. Prove that if M is a k-state synchronizable DFA, then it has a synchronizing sequence of length at most $k^3$. Can you improve upon this bound?

## Step-by-step solution

**Step 1** of 1

**Given:**

A DFA $M = (Q, \Sigma, \delta, q_0, F)$ and suppose $h$ is a state of DFA $M$ known as its home state.

**Proof:**

The at most length of synchronizing sequences is $k^3 0$ for a $k$-state synchronizable DFA. In the year 1964, a Slovak scientist named Jan Cerny first tried to solve the problem of synchronizing automata in real time. This problem is sometimes referred as *Cerny's Conjecture.*

To prove the upper bound on the synchronizing sequence we try to device a greedy algorithm.

**Algorithm:**

1. Let a synchronizing DFA be $M = (Q, \Sigma, \delta, q_0, F)$. Initialize the synchronizing sequence $\omega \leftarrow 1$ (empty word) and a set of states $P \leftarrow Q$.

2. ***while*** $|P| > 1$

a. Find a word $\upsilon$ which is belongs to $\Sigma^*$ and it has minimum length $|\delta(P, \upsilon)| < |P|$.

b. If none exists, ***return*** failure.

c. $\omega \leftarrow \omega\upsilon$

$P \leftarrow \delta(P, \upsilon)$

3. ***return*** $\omega$

• Now suppose that $M$ is a $k$-state DFA, that is, $|Q| = k$ then clearly the main loop of the algorithm runs at most $k-1$ times. In order to get the length of the output word $\omega$ user has to estimate the length of each word $\upsilon$ derived at each loop.

• Consider a generic step at which $|P| = n > 1$ and let $\upsilon = a_1 \cdots a_l$ with $a_i \in \Sigma,\ i = 1 \cdots l$. Then it is quite simple to see that the sets, $P_1 = P,\ P_2 = \delta(P_1, a_1),\ \ldots,\ P_l = \delta(P_{l-1}, a_{l-1})$ are $n$-element subsets of $Q$.

• Furthermore, since $|\delta(P_l, a_l)| < |P_l|$, there exists two states $q_l, q_l' \in P_l$ such that $\delta(q_l, a_l) = \delta(q_l', a_l)$.

• Now define two element subsets $R_i = \{q_i, q_i'\} \subseteq P_i,\ i = 1, \cdots, l-1$. then the condition that $\upsilon$ is a word that has minimum length $|\delta(P, \upsilon)| < |P|$ which implies that $R_i \not\subset P_j$ for $1 \le j < i < l$. Now by the Peter Frankl inequality, the tight bound over $l$ be $\binom{k-n+2}{2}$.

• Summing up these inequalities from $n = k$ to $n = 2$, user can get the upper bound over the synchronizing sequence $|\omega| \le \frac{k^3 - k}{6}$.

**Conclusion:**

Therefore, at most length of synchronizing sequences is $k^3 0$ for a $k$-state synchronizable DFA.

Comment

## Problem

Let $\Sigma = \{a, b\}$. For each $k \geq 1$, let $C_k$ be the language consisting of all strings that contain an a exactly k places from the right-hand end. Thus $C_k = \Sigma^* a \Sigma^{k-1}$. Describe an NFA with k + 1 states that recognizes $C_k$ in terms of both a state diagram and a formal description.

## Step-by-step solution

**Step 1** of 2

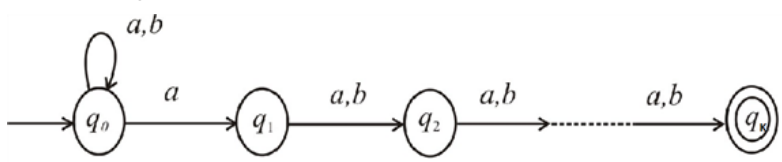
Given language is

$C_k = \Sigma * a\ \Sigma^{k-1}$ for each $K \geq 1$, over the alphabet $\Sigma = \{a, b\}$

$C_k$ is the language consisting of all strings that contains an '*a*' exactly *K* places from the right – hand end.

Let *N* be the NFA with $K+1$ states that recognizes $C_K$

(i) The state diagram of NFA *N* is follows:




Comment

**Step 2** of 2

(ii) The formal description of NFA *N* is as follows:

$N = (Q, \Sigma, \delta, q_0, F)$

$Q =$ set of sates $= \{q_0, q_1 \ldots, q_K\}$

$\Sigma =$ set of alphabet $= \{a, b\}$

$q_0 =$ start state $= \{q_0\}$

$F =$ set of final states $= \{q_K\}$

$\delta =$ The transition function is given as follows:

$$\delta(q_i, a) = \begin{cases} \{q_0, q_1\} & \text{if } i = 0 \\ \{q_{i+1}\} & \text{if } 0 < i < k \\ \phi & \text{if } i = k \end{cases}$$

$$\delta(q_i, b) = \begin{cases} \{q_0\} & \text{if } i = 0 \\ \{q_{i+1}\} & \text{if } 0 < i < k \\ \phi & \text{if } i = k \end{cases}$$

$\delta(q_i, \in) = \phi \forall i.$

Comment

## Problem

Consider the languages $C_k$ defined in Problem 1.60. Prove that for each k, no DFA can recognize $C_k$ with fewer than $2_k$ states.

## Step-by-step solution

**Step 1** of 2

Consider the language $C_k = \Sigma * a\Sigma^{k-1}$ for each $k \geq 1$, over the alphabet $\Sigma = \{a,b\}$. $C_k$ be the language consisting of all strings that contain an '*a*' exactly $k$ places from the right-hand end.

Comment

**Step 2** of 2

Now it is required to prove that, no DFA (Deterministic finite automation) can recognize $C_k$ with fewer than $2^k$ states.

If a DFA enters two different states after reading two different strings *lz* and *mz* with same suffix *z*, then the DFA enters two different states after reading the strings *l* and *m*. Take two different strings of length *k* such that $l = l_1 \ldots . l_k$ and $m = m_1 \ldots . m_k$. Let *i* be some position such that $l_i \neq m_i$. One of the strings contains *a* in its $i^{th}$ position and the other string contains *b* in its $i^{th}$ position.

Consider the suffix string $z = b^{i-1}$. In this case, either the string *lz* or *mz* has the $k^{th}$ bit from the end as *a*. The total number of strings of length *k* over the input alphabet $\{a,b\}$ is $2^k$. Thus, the DFA needs $2^k$ states in order to distinguish $2^k$ strings.

Therefore, the DFA that recognizes the language $C_k$ has at least $2^k$ states.

Comment

## Problem

Let ∑ = {a, b}. For each k ≥ 1, let $D_k$ be the language consisting of all strings that have at least one a among the last k symbols. Thus $D_k = \Sigma^* a(\Sigma \cup \varepsilon)^{k-1}$. Describe a DFA with at most k+1 states that recognizes$D_k$ in terms of both a state diagram and a formal description.

## Step-by-step solution

**Step 1** of 2

The Language $D_k = \Sigma^* a(\Sigma \cup \varepsilon)^{k-1}$ for each $k \geq 1$, over the alphabet $\Sigma = \{a, b\}$. Here, $D_k$ is the language consisting of all strings that have at least one $a$ among the last $k$ symbols. The DFA (deterministic finite automata) that recognizes the language $D_k$.

The construction of $M$ where $M = (Q_k, \Sigma, \delta_k, q_0, F)$ is as follows:

- $Q_k = \{q_0, q_1, q_2, \ldots q_k\}$ = set of states
- $\Sigma = \{a, b\}$ = set of alphabets
- $q_0 = \{q_0\}$ = start state.
- $F = \{q_1, q_2 \ldots q_k\}$ = set of final states
- $\delta_k =$ Transition function is given as follows

$$\delta_k(q, l) = \begin{cases} q_1 & i = 0 \wedge l = a \\ q_0 & i = 0 \wedge l = b \\ q_1 & i \neq 0 \wedge l = a \\ q_{(i+1) \bmod k} & i \neq 0 \wedge l = b \end{cases}$$

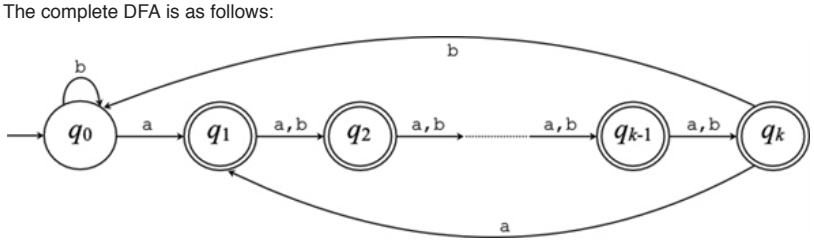
Comment

**Step 2** of 2

In other words,

- $M$ is in state $q_0$ if it has not seen and $a$ within the past $k$ letters then it will reject.
- It is in state $q_i$ if it saw $a$ from $i$ letters ago.

The complete DFA is as follows:




Comment

## Problem

**a.** Let A be an infinite regular language. Prove that A can be split into two infinite disjoint regular subsets.

**b.** Let B and D be two languages. Write $B \Subset D$ if $B \subseteq D$ and $D$

## Step-by-step solution

**Step 1** of 2

**a.**

The proof that an infinite regular language, say A, can be split into two infinite disjoint regular subsets is as follows:

• Let there be a string, say 's', such that $s \in A$ and s = xyz, where x, y and, z, represent the sub-strings of the string s.

• Since s belongs to the language A, and the language A is regular, $xy^iz$ must belong to A, where $i \geq 0$ . (As per the condition 1 of pumping lemma).

• Let $A_1$ be a language such that $A_1 = \{xy^{2i}z, \text{where } i \geq 0\}$.

• Since all the strings of the form $xy^iz$ belong to A, the strings of the form $xy^{2i}z$ must also belong to A.

• Hence, the language $A_1$ is a subset of the language A, i.e:

$A_1 \subset A$

• The strings of the language $A_1$ can be represented by the following regular expression:

$x(yy)^*z$

Hence, the language $A_1$ is a regular language

• Since in the expression, $A_1 = \{xy^{2i}z, \text{where } i \geq 0\}$, there is no upper limit for the value of i, the language $A_1$ is infinite.

• Since the regular languages are closed under the operation of complement, the language $\overline{A_1}$ is a regular language.

• Let $A_2$ be a language such that, $A_2 = \overline{A_1} \cap A$.

• Since the regular languages are closed under the operation of intersection, the language $A_2$ is a regular language.

• Since the languages, $A_1$ and A are infinite, the language A2 is also infinite

• Clearly $A_2$ and $A_1$ are two disjoint sets.

• Also, $A = A_1 \cup A_2$

Thus, the language A can be split into two infinite disjoint regular subsets.

**Hence, proved.**

Comment

**Step 2** of 2

**b.**

The steps required to prove the given statement are as follows:

• Divide the regular language D into two regular disjoint subsets and let one of those subsets be B.

• Let the other subset be A, such that A = D – B.

• Since D contains infinitely many strings that are not in B, A also contains infinitely many strings that are not in B.

• Further divide the language A into two disjoint subsets, $A_1$ and $A_2$, such that $A_2$ contains infinitely many strings that are not in $A_1$ and vice versa.

- Since A contains infinitely many strings that are not present in B, $A_1$ also contains infinitely many strings that are not in B.
- Create a set C such that $C = A_1 \cup B$ .
- Since $A_1$ contains infinitely many strings that are not in B, C also contains infinitely many strings that are not in B.
- Clearly, B is a subset of C.
- Hence, the following statement is true:

$B \Subset C$

- Since $A_2$ contains infinitely many strings that are not in $A_1$, D contains infinitely many strings that are not present in $A_1$.
- Since D contains infinitely many strings that are not present in $A_1$, D contains infinitely many strings that are not present in C.
- Hence, the following statement is true:

$C \Subset D$.

- Since $B \Subset C$ and $C \Subset D$, the following statement is true:

$B \Subset C \Subset D$

**Hence, proved.**

Comment

## Problem

Let N be an NFA with k states that recognizes some language A.

**a.** Show that if A is nonempty, A contains some string of length at most k.

**b.** Show, by giving an example, that part (a) is not necessarily true if you replace both A's by $\overline{A}$.

**c.** Show that if $\overline{A}$ contains some string of length at most $2^k$.

**d.** Show that the bound given in part (c) is nearly tight; that is, for each k, demonstrate an NFA recognizing a language $A_k$ where $\overline{A_k}$ shortest member strings are of length exponential in k. Come as close to the bound in (c) as you can.

## Step-by-step solution

**Step 1** of 5

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA with *K* states that recognizes some language *A*

(a) Suppose *A* is non empty

• Then there must be an accept state $q \in F$ that can be reached from the start state $q_0$.

• Let *w* be the string that can be accepted by *N* when traveling along the shortest path $q_0$ to $q$.

• Let *n* be the length of *w*.

• Then, the sequence of state $q_0, q_1 ... q$ in the shortest path from $q_0$ to *q* has length $n+1$.

• Note that all the states from $q_1$ to $q$ in this sequence must be distinct; otherwise we would find a shorter path from $q_0$ to $q$ by removing the repeated states.

• Since there are only *K* states in N and there are *n* distinct states in the shortest path from $q_0$ to $q$, we have $n \le K$.

• Clearly, *w* is accepted by *N* because *q* is an accept state

• So *A* contains a string of length at most *K*.

Comment

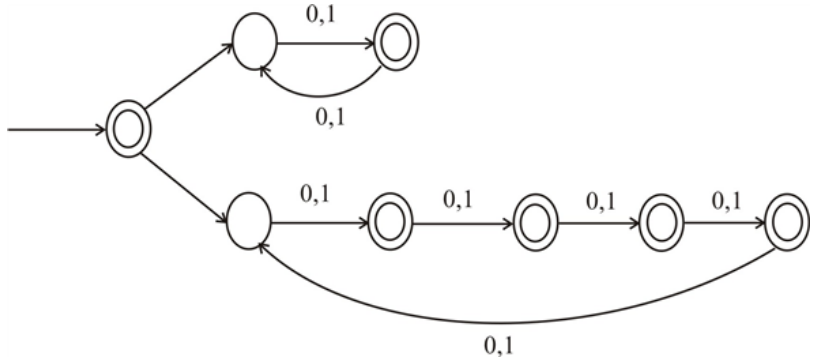
**Step 2** of 5

(b) Example:

Suppose $\Sigma = \{0,1\}$ and N be the NFA with $K = \delta$ states.

Let *A* be the language recognized by *N*.

The State diagram of *N* is as follows




Comment

**Step 3** of 5

• Clearly *N* accepts the empty string.

• For any nonempty string *w*, *N* will reject *w* if and only if the length of *w* is divisible by 2 and 5.

• Thus $\overline{A}$ consists of all non empty strings of length divisible by 10

• So $\overline{A}$ is non-empty and the shortest string in $\overline{A}$ has length $10 > K$

• Hence we got the contradiction of part (a) when we replace *A* by $\overline{A}$.

Comment

**Step 4** of 5

(c) We know that " Every non deterministic finite automaton has an equivalent deterministic finite automaton"

• So we convert *N* into a DFA *M* that also recognizes *A*, where the set of states in *M* is the set of subsets of *Q*.

• Then we swap the accept and non accept states of *M* to obtain a DFA $\overline{M}$ that recognizes $\overline{A}$.

• Note that $\overline{M}$ has $2^K$ states.

• Applying part (a) by replacing *N* with $\overline{M}$, we can conclude that if $\overline{A}$ is non-empties, then $\overline{A}$ contains a string of length at most $2^K$.

Comment

**Step 5** of 5

(d) The idea used in part (b) can be generalized to obtain a bound close to an exponential form.

• Let $2 \le P_1 < P_2 < ... < P_n \le \sqrt{K}$ be all the primes in $\left[1, \sqrt{K}\right]$

• For each $P_i$, construct a DFA $M_i$ of $P_i$ states that rejects only strings of length divisible of $P_i$.

• Finally, construct an NFA *M* by union all these $M_i$ machines in the following ways:

→ create a separate starting state $q_0$ add an $\in$ transition from $q_0$ to the starting states of each $M_i$, and also designate $q_0$ as an accepting state.

→ This machine M will have $1 + P_1 + P_2 + ... + P_m \le K$ states.

→ On the other hand, *M* rejects a string *w* if and only if a is non empty and $|w|$ is divisible by $P_1 P_2 ... P_m$.

→Hence, the shortest string is rejected by *M* has length $P_1 P_2 ... P_m$.

→ Now coming to the analysis of the bound,

By the prime Number theorem, there are approximately $\ln$ *n* prime numbers in $[0, n]$ for *n* sufficiently large.

→ Hence, $m \approx \frac{1}{2} \ln K$.

→ Since there are about $\frac{1}{4} \ln K$ primes in $\left[0, \sqrt[4]{K}\right]$, there are about $m - \frac{1}{4} \ln K \approx \frac{1}{4} \ln K$ primes in $\left[\sqrt[4]{K}, \sqrt{K}\right]$

Thus $P_1 P_2 ... P_m \ge \left(\sqrt[4]{K}\right)^{\frac{1}{4} \ln K} = K^{\frac{1}{16} \ln K}$

## Problem

Prove that for each n > 0, a language $B_n$ exists where

**a.** $B_n$ is recognizable by an NFA that has n states, and

**b.** if $B_n = A_1 \cup \cdots \cup Ak$, for regular languages $A_i$, then at least one of the $A_i$ requires a DFA with exponentially many states.

## Step-by-step solution

**Step 1** of 2

a.

**Proving that a language is recognizable by an NFA**

Suppose $B_n$ be a language where $n > 0$. User has to prove that the language is recognizable by an NFA with $n$ states.

**BASIS:** Let $n = 1$ hence $B_n = \{\varepsilon, 0, 1\}$. Therefore formally we can design an NFA $N = (\{q_o\}, \Sigma, \delta, q_0, \{q_0\})$ with a single state that accepts all the given language as $\delta(q_0, \varepsilon \mid 0 \mid 1) = q_0$.

**Proof by induction:** suppose one can divide $B_n$ in two regular expressions say $E$ and $F$ of length $n_1, n_2 < n$ and $n_1 + n_2 = n$.

Now by inductive hypothesis it can easily concluded that the NFA's accepting $E$ and $F$ are consisting of at least $n_1$ and $n_2$ states.

But it is already known to us that the set of regular expression is closure under Union, Concatenation and Star operation.

Therefore the language $B_n$ is recognizable by an NFA with $n$ states.

Comment

**Step 2** of 2

From the above part it is proved that a language $B_n$ where $n > 0$ is recognizable by an NFA with $n$ states.

Now for $B_n = A_1 \bigcup A_2 \bigcup \cdots \bigcup A_k$ where $A_i$'s are regular.

If a DFA is constructed which is equivalent to the DFA of the given NFA.

There could be at least $n$ and at most $2^n$ states in the resultant equivalent DFA. Every regular language is recognized by a DFA so there is a corresponding DFA for all the $A_i$s. Now, by the pigeon hole principle, one can state that there is at least one DFA which requires $2^i$ states to recognize a language among all the $A_i$.

Comment

## Problem

A ***homomorphism*** is a function $f: \Sigma \longrightarrow \Gamma^*$ for any language A.

**a.** Show, by giving a formal construction, that the class of regular languages is closed under homomorphism. In other words, given a DFA M that recognizes B and a homomorphism f, construct a finite automaton M` that recognizes f(B). Consider the machine M` that you constructed. Is it a DFA in every case?

**b.** Show, by giving an example, that the class of non-regular languages is not closed under homomorphism.

## Step-by-step solution

**Step 1** of 2

a.

**Class of Regular Languages is closed under Homomorphism:**

In general by the definition of homomorphism if $R$ be a regular expression where each symbol $a$ in $\Sigma$, Assume, $f(R)$ be the expression this is obtained by replacing each symbol $a$ in $R$ by $f(a)$.

Now assume a language $B = B(R)$ for some regular expression $R$. Here $f(R)$ defines the language $f(B)$.

Therefore it is needed to proof that if user take a sub-expression $E$ of $R$ and apply homomorphism $f$ to it and get $f(E)$, the language of $f(E)$ is the same language if user apply $f$ to the language $L(E)$. It implies $L(f(E)) = f(L(E))$.

**BASIS:** Suppose, $E = \Phi$ that is $L(E)$ does not contain any strings, then $f(L(E)) = L(f(E))$. Again suppose $E = \{\varepsilon\}$ that is $L(E)$ contains a string with no symbols, then also $f(L(E)) = L(f(E))$

Now assume $E = \{a\}$ for some symbol $a \in \Sigma$. Here, in this case $L(E) = \{a\}$ then $f(L(E)) = \{f(a)\}$. Again, $f(E)$ is the regular expression that is the string of symbols $f(a)$,

Therefore, it can be said that $L(f(E)) = \{f(a)\}$.

It implies $L(f(E)) = f(L(E))$.

**INDUCTION:** By the rule of union over regular expression. Assume, $E = E_1 + E_2$.

By the application of homomorphism over regular expression user may say that, $f(E) = f(E_1) + f(E_2)$. By the grammar of regular language $L(E) = L(E_1) \cup L(E_2)$.

Therefore,

$$\begin{aligned} L(f(E)) &= L(f(E_1) + f(E_2)) \\ &= L(f(E_1)) \cup L(f(E_2)) \end{aligned}$$

Again, though $f$ is applied to all the strings of a language separately and individually,

$$\begin{aligned} f(L(E)) &= f(L(E_1) \cup L(E_2)) \\ &= L(f(E_1)) + L(f(E_2)) \end{aligned}$$

Here by the Inductive Hypothesis we may assert that, $L(f(E_1)) = f(L(E_1))$ and $L(f(E_2)) = f(L(E_2))$. It implies $L(f(E)) = f(L(E))$.

Hence, it can be said that the class of regular languages is closed under homomorphism.

Comment

**Class of non-regular language is not assumed to be closed under homomorphism:**

Consider a given language $B=\left\{0^n1^n \middle| n\geq 0\right\}$ is defined over the alphabet $\Sigma=\{0,1\}$. Now by pumping lemma of regular languages it can easily be said that the language $B$ is non-regular.

Here another alphabet set is $\Gamma=\{a,b\}$ also defined.

The homomorphism function $f:\Sigma\to\Gamma^*$ is defined as $f(0)=ab$ and $f(1)=\varepsilon$. Therefore the language $B'$ over alphabet $\Gamma$ could be described as $B'=\left\{(ab)^n \middle| n\geq 0\right\}$.

If user apply pumping lemma of regular languages over $B'$ and divide it in $xyz$ where $x$ is $\varepsilon$, $y=ab$ and $z$ is also $\varepsilon$ then we can say that $B'$ is a regular language.

Therefore user can see from this that applying homomorphism over a non-regular language yields a regular language.

Hence homomorphism is not closed over the class of non-regular languages.

Comment

## Problem

Let the ***rotational closure*** of language $A$ be $RC(A) = \{yx \mid xy \in A\}$.

**a.** Show that for any language A, we have *RC*(A) = *RC*(*RC*(A)).

**b.** Show that the class of regular languages is closed under rotational closure.

## Step-by-step solution

**Step 1** of 4

a) The Rotational Closure of a language A is defined as $RC(A) = \{yx | xy \in A\}$. Now for any language $A$ a string $\omega \in A$ also $\omega \in \Sigma^*$, and therefore $\varepsilon \in A$.

It is known that $\omega\varepsilon = \varepsilon\omega = \omega$. Here for any language $A = \{\omega\varepsilon\}$, $RC(\omega\varepsilon) = \varepsilon\omega = \omega$ and $RC(RC(\omega\varepsilon)) = RC(\varepsilon\omega) = \omega\varepsilon = \omega$.

Therefore, $\boxed{RC(A) = RC(RC(A))}$

Comments (1)

**Step 2** of 4

b) Proof: Assume the language $A$ is being defined with the help of regular expression $E$. Now structural induction on the size of regular expression $E$ is to be proved.

Now, it is to be shown that $A(E) = A(RC(A))$, which implies the language $RC(A)$ is the reverse language of the specified language $A$.

Basis: If $E$ is $\varepsilon, \varnothing$, or $a$ for some symbol $a$, then $RC(E) = E$. That is, $RC(\varepsilon) = \varepsilon$, $RC(\varnothing) = \varnothing$ and $RC(a) = a$.

Induction: Three cases arise depending upon the specified expressions $E$ which are as follows:

- $E = E_1 + E_2$. Then $RC(E) = RC(E_1) + RC(E_2)$.

The relational closure of any two languages is obtained with the computation and after that taking the union of the specified two languages.

Comment

**Step 3** of 4

- $E = E_1E_2$. Then $RC(E) = RC(E_2)RC(E_1)$.

Here the expression is rotated with respect to two languages and language itself

For an example let if, $L(E_1) = \{01,111\}$ and $A(E_2) = \{00,10\}$, then,

$$A(E_1E_2) = \{0100, 0110, 11100, 11110\}, \text{ then we can say}$$
$$RC(A(E_1E_2)) = \{0010, 0110, 00111, 01111\}, \text{ where } \varepsilon \in A.$$
$$\text{Again, } RC(A(E_1)) = \{10, 111\}$$
$$RC(A(E_2)) = \{00, 01\}$$
$$A(RC(E_2)RC(E_1)) = \{0010, 00111, 0110, 01111\}$$
$$\text{and therefore, } RC(A(E)) = A(RC(E)).$$

Comment

**Step 4** of 4

- $E = E_1^*$. Then $RC(E) = (RC(E_1))^*$. Here the justification is any string $\omega$ in $A(E)$ can be written as $\omega_1\omega_2\cdots\omega_n$ where $\omega_i \in L(E)$.

But $RC(\omega) = RC(\omega_n)RC(\omega_{n-1})\cdots RC(\omega_1)$ where each $RC(\omega_i) \in A(RC(E))$ and hence $RC(\omega)$ is in $L\left((RC(E_1))^*\right)$.

Therefore it is proved that the set of regular languages are closed under rotational closure.

Comment

## Problem

In the traditional method for cutting a deck of playing cards, the deck is arbitrarily split two parts, which are exchanged before reassembling the deck. In a more complex cut, called Scarne's cut, the deck is broken into three parts and the middle part in placed first in the reassembly. We'll take Scarne's cut as the inspiration for an operation on languages. For a language A, let *CUT*(A) = {yxzl xyz ? A}.

**a.** Exhibit a language B for which *CUT*(B) ≠ *CUT*(*CUT*(B)).

**b.** Show that the class of regular languages is closed under *CUT*.

## Step-by-step solution

**Step 1** of 2

**Consider the following language:**

$B = \{0^n 1^n \mid \{0,1\} \in \Sigma, n \geq 0\}$.

The above language $B$ is a language for which $CUT(B) \neq CUT(CUT(B))$. In the above language, if the string $w$ exists in $B$ and user divides this string $w$ in $xyz$ as $0^n 1^m 1^{n-m}$ then $CUT(B) = 1^m 0^n 1^{n-m}$. After again dividing it becomes $CUT(CUT(B)) = 0^n 1^n$.

Hence $CUT(B) \neq CUT(CUT(B))$

Comment

**Step 2** of 2

Assume that a regular language $A$ and prove that $CUT(A)$ is also a regular language. Again, let the DFA that accepts $A$ be $M_A$.

Now construct another DFA $M_{CUT}$ that accepts the language $CUT(A)$ and hence it will be proved that $CUT(A)$ is a regular language.

**DFA Construction:** Let $M = (Q, \Sigma, \delta, q_0, F)$ recognizes $A$. Construct $M_{CUT} = (Q_{CUT}, \Sigma, \delta_{CUT}, q_{CUT}, F_{CUT})$ to recognize $CUT(A)$.

i) $Q_{CUT} = Q$. The states of $M_{CUT}$ are same as $M$.

ii) The new start state of the DFA is $q_{CUT}$.

iii) $F_{CUT} = F$. Let $w$ be a string in $A$ accepted by $M$. If user divides this string $w$ in $xyz$ then $w'$ in $CUT(A)$ be $yxz$ which will be accepted by $M_{CUT}$. Now in both cases the machine enters into a accept state on input of the substring $z$. Hence the accepting states of both of the machines are same.

iv) Now define transition function of $M_{CUT}$, $\delta_{CUT}$ for $q_{CUT} \in Q_{CUT}$ and $a \in \Sigma$ as follows:

$$\begin{aligned}\delta_{CUT}(q_{CUT}, a) &= \delta(q, a) \text{ where } q \in Q \text{ and } q \notin F \\ &= \delta(q, a) \text{ where } q \in F\end{aligned}$$

Hence, from the above DFA it is clear that accepting states of both of the machines are same. So, the class of regular languages is closed under $CUT$ operation.

Comment

## Problem

Let $\Sigma = \{0,1\}$. Let $WW_k = \{ww \mid w \in \Sigma^*$ and w is of length k}.

**a.** Show that for each k, no DFA can recognize $WW_k$ with fewer than $2^k$ states.

**b.** Describe a much smaller NFA for $\overline{WW_k}$, the complement of $WW_k$.

## Step-by-step solution

**Step 1** of 2

**Consider the following language:**

$WW_k = \{ww \mid w \in \Sigma^* \text{ and } w \text{ is of length } k\}$.

Therefore a string in $WW_k$ language is at least of length $k$. Now consider two different $k$- bit strings $x = x_1 \ldots x_k$ and $y = y_1 \ldots y_k$, $i$ be some location such that $x_i \neq y_i$.

Hence one of the strings contains a 1 at the $i^{th}$ position where other contains a 0. Assume that $z = 0^{i-1}$.

Then $z$ distinguishes $x$ and $y$ as exactly one of $xz$ and $yz$ has the $k^{th}$ bit from the end as 1.

Since there exists, total $2^k$ binary strings of length $k$ which can be mutually distinguish by the argument mentioned above, so any DFA for the given language has at least $2^k$ states.

Comment

**Step 2** of 2

**The language $\overline{WW_k}$ can be described as follows:**

$\overline{WW_k} = \{ww \mid w \in \Sigma^* \text{ and } |w| < k\}$.

Therefore the user can build non deterministic finite automata with exactly $k+1$ states. This NFA can recognize the language $\overline{WW_k}$.

Assume that the non-deterministic finite automata consist of $Q = \{0,1,\ldots,k\}$ with the names of the state's corresponding to how many of the last $k$ bits the NFA has seen.

Define $\delta(0,0) = 0, \delta(0,1) = 1$ and $\delta(i-1,0 \mid 1) = i$ for $2 \le \mathrm{i} \le \mathrm{k}$. We set $q_0 = 0$ and $F = \{k\}$

The machine starts from the initial state which is state 0, as it will traverse 1 initial state which is state 0, as it will traverse 1 it understand that it is the $k^{th}$ bit from the finishing state or end state and proceed to state 1.

As it has reached on the state $k$, it accepts the string if and only if the string contains exactly $k-1$ bits. It checks for $k-1$ bits as it starts traversing from $0^{th}$ index.

Comments (5)

## Problem

We define the ***avoids*** operation for languages A and B to be

A *avoids* B = {wl w ? A and w doesn't contain any string in B as a substring}.

Prove that the class of regular languages is closed under the *avoids* operation.

## Step-by-step solution

**Step 1** of 3

**Regular Language:**

• Regular language is the language which is generated or expressed with the help of regular expression and is recognizable by the finite automata.

• It is possible to get different types of regular languages from different types of regular grammar.

• By the definition of regular language it is known that if a finite automata recognizes a language then only it is said to be regular.

• Thus, in this case of **"avoids"** operation over two regular languages A and B

• If a finite automata accepting the language of **"avoids"** can be constructed

• Then it can be said that the set of regular languages is closed under the operation **"avoids".**

Comment

**Step 2** of 3

**As per given details there are two languages as *A* and *B* for which the avoid operations is defines as follows:**

$$\text{A avoids B}=\{\text{w}\,|\,\text{w} \in \text{A and w does not contain any string in B as a substring}\}.$$

Here, it is required to prove that class of regular languages is closed under avoids operation.

Comment

**Step 3** of 3

**Proof:**

• In the above:

(A avoids B) can be written as follows:

$$(\text{A} - (\text{A has B}))$$

Where (A has B) are string of A which contain strings of B as substrings.

• After that $(\text{A avoids B})$ will be strings of A that means strings of B will not contain B as substring.

• (A has B) can be written as follows:

$$(\text{A has B}) \text{ as } \text{A} \cap (\Sigma^* \text{B} \Sigma^*)$$

• After that $(\text{A has B})$ that means these are the strings of A that contains the strings of B as substring.

• Similarly, (A avoids B) can be written as follows:

$$(\text{A} - (\text{A} \cap (\Sigma^* \text{B} \Sigma^*)))$$

**Hence, the regular languages are closed under the concatenation, intersection, and subtraction and even closed under operation avoid.**

Comments (1)

## Problem

Let ∑ = {0,1}.

**a.** Let $A = \{0^k u 0^k \mid k \geq 1 \text{ and } u \in \Sigma^*\}$. Show that $A$ is regular.

**b.** Let $B = \{0^k 1 u 0^k \mid k \geq 1 \text{ and } u \in \Sigma^*\}$. Show that $B$ is not regular.

## Step-by-step solution

**Step 1** of 6

**Pumping Lemma:** There is an integer $i$ for any language $L$ which is regular in such a way that $X$ belongs to $L$ with $|X| \geq i$, There exits $p,q,r \in \sum^*$, such that $X = pqr$, and(a) $|pq| \leq i$ (b) $|q| > 0$ (c) $\forall n \geq 0 : pqr \varepsilon L$

Comment

**Step 2** of 6

**(a)**

**Consider the following details which is as follows:**

Given language $A = \{0^k u\, 0^k \mid k \geq 1 \text{ and } u \in \Sigma^*\}$ and $\Sigma = \{0,1\}$

- In order to prove whether the given language is regular, the principle of pumping lemma is applied.
- Now, applying pumping lemma of regular language on the language $A$ and let $p$ be the minimum pumping length.
- Consider a string $\omega = 0^k u 0^k$ where $\omega$ is in language $A$ of length $p$ where p= $p = (2k + |u|)$
- Now, if $\omega$ is divided in $xyz$ and it is known that $|xy| \leq p$ for $x = 0^k, y = u, z = 0^k$.

Comments (3)

**Step 3** of 6

**Case 1:**

- If $u = \xi$
- Then language must be $0^k 0^k \in A$

$$\omega = xy^i z$$
$$let\, i = 0$$
$$then\, \omega = 0^k \xi 0^k$$
$$= 0^k 0^k \in A$$

**Hence, for i=0 and $\omega$ is in specified language.**

Comment

**Step 4** of 6

**Case 2:**

$\omega = xyz$

$then\, x = z = 0^k$

$(|x| + |y| + |z|) <= p$

$k + |y| + k <= p$

$then |y| <= p - 2k$

• Let y=1 then the language will be:

$0^k 1 0^k \in A$

**Hence, for y=1 and $\omega$ is in specified language.**

**Hence for any y belonging to $\sum^*$ . So $\omega$ is in specified language.**

Comment

**Step 5** of 6

**Case 3:**

$\omega = xy^{p-2k}z$

$\omega = 0^k y^{p-2k} 0^k$

**So, the** $y^{p-2k}$ always belongs to $\sum^*$

• So, from the pumping lemma it is proved that the language is regular as $\omega$ always belongs to specified language.

Comments (2)

**Step 6** of 6

**(b)**

**Consider the following details which is as follows:**

Given language $B = \left\{0^k 1 u\, 0^k \middle| k \geq 1 \text{ and } u \in \Sigma^*\right\}$ and $\Sigma = \{0,1\}$

• In order to prove whether the given language is regular, the principle of pumping lemma is applied.

• Now, applying pumping lemma of regular language on the language $B$ and let $p$ be the minimum pumping length.

• Consider a string $\omega = 0^2 1 u 0^2$ where $\omega$ is in language $B$ of length *p* for k=2.

• Now, if $\omega$ is divided in $xyz$ and it is known that $|xy| \leq p$ for x=0, y= $0$ and z=$1u0^2$

• Apply condition 1 in pumping lemma that is $xy^i z \in B$ for $i \geq 0$.

• Assume that i=2, so x=0 $y^2 = 00$ z=1 $u0^2$

• Now $xy^i z$ is 0001u$0^2$ and can be written as $0^3$1u$0^2$

• It is clear that $0^3$1u$0^2$ does not belong to language *B* because the value of k is not same for string $0^k 1 u\, 0^k$ | k≥1.

• So { $xy^i z = 0^3 1 u 0^2$ for i=2} $\notin$ *B*.

**Hence, it is proved that *B* is a not a regular language.**

Comments (2)

## Problem

Let $M_1$ and $M_2$ be DFAs that have $k_1$ and $k_2$ states, respectively, and then let U = L($M_1$) [ L($M_2$).

**a.** Show that if $U \neq \emptyset$, then $U$ contains some string $s$, where $|s| < \max(k_1, k_2)$.

**b.** Show that if $U \neq \Sigma^*$, then $U$ excludes some string $s$, where $|s| < k_1 k_2$.

## Step-by-step solution

**Step 1** of 4

**a)**

Suppose $M_1$ and $M_2$ are taken as DFAs which consists $k_1$ and $k_2$ states respectively. Now, consider $U = L(M_1) \cup L(M_2)$. So, it can be shown that "if $U \neq \phi$, then some string $s$ exists in $U$, where $|s| < \max(k_1, k_2)$".

Comment

**Step 2** of 4

To prove the above statement first assume that the DFAs $M_1$ and $M_2$, which are taken above, consists strings of unequal length and also different from each other.

- Now consider a start state, for the given grammar $U$, $q$ is taken. Then, the production of $M_1$ and $M_2$ with a start variable $q$ is given by:
  $q \to sM_1 \mid sM_2$
- The string consists of unequal length and also different from each other and the start variable are initiated for the grammar.
- Then, from the property of union, it will take the longest string from the taken two strings and it will add the un-matched string or variable.

Therefore, the final string which is taken should be less than the any string taken. Thus it can be said that **"if $U \neq \phi$, then some string $s$ exists in $U$, where $|s| < \max(k_1, k_2)$".**

Comment

**Step 3** of 4

**b)**

Suppose $M_1$ and $M_2$ are taken as a DFAs which consists $k_1$ and $k_2$ states respectively. Now, consider $U = L(M_1) \cup L(M_2)$. So, it can be shown that" if $U \neq \Sigma^*$, then some string $s$ can be excluded from $U$, where $|s| < k_1 k_2$".

Comment

To prove the above statement first assume that the DFAs $M_1$ and $M_2$, which are taken above, consists strings of unequal length and also different from each other.

- Now consider $U = L(M_1) \cup L(M_2)$, which says that $U$ consists the string of $M_1$ and $M_2$. The string which is taken may be longer in length than $M_1$ and $M_2$.
- Here, concatenation operation is applied between the initial string $k_1$ and $k_2$, then the length it consists may be longer than the original length of the string.
- Some string also exists in this final concatenated string, which is already taken. Therefore, these strings will be excluded in the final string taken.

Hence from the above discussion it can be said that "**if $U \neq \Sigma^*$, then some string $s$ can be excluded from $U$, where $|s| < k_1 k_2$**".

Comment

## Problem

Let

$\Sigma = \{0,1,\#\}$. Let $C = \{x\#x^{\mathcal{R}}\#x \mid x \in \{0,1\}^*\}$. Show that $\overline{C}$ is a CFL.

## Step-by-step solution

**Step 1** of 4

Suppose $\Sigma = \{0,1,\#\}$. Now consider a language $C$, where $C = \left\{x\#x^R\#x \mid x \in \{0,1\}^*\right\}$, then it has to show that $\overline{C}$ **is a context free language**. This can be achieved by showing the contradiction, that is, the language $C$ is not a context free language.

Comment

**Step 2** of 4

Each string $k = x\#x^R\#x \in C$, contains $|x| = |x^R|$, and then $|k|$ is multiple of three. For a contradiction suppose that $C$ is context free. As it is considered as a context free, so it contains a pumping length $l$. Take $k = 0^{2l}0^l1^l0^{2l} \in C$ with $|k| > l$. Hence, there exists $abcde$ in such a way that:

1. $ab^m cd^m e \in A$, for all $m \geq 0$.
2. $|bd| > 0$. It means that either b or d should not be empty string.
3. $|bcd| \leq l$. It means that length of the parts b, c, and d should be at most p.

Now, the following cases, which are given below, show that "the value taken for $abcde$ is independent and always **shows a contradiction**".

Comment

**Step 3** of 4

Consider the different **cases** which are given below:

**Case 1:** $|bd|$ is not multiple of three. Then, $k' = ab^2cd^2e \notin C$ since $|k'|$ will no longer be multiple of three.

**Case 2:** Only 0s from the prefix set of 0s exists in $bcd$ and $|bd| = 3q$ for some $q$. Then, $ab^2cd^2e = 0^{3l+3q}1^l0^{2l} = 0^{2l+q}0^{l+2q}1^{l-q}1^q0^{2l} \notin C$. Since $x = 0^{2l+q}$ and $x^R \neq 1^q0^{2l}$, the **resultant third of the string**.

**Case 3:** $bcd$ has only 1s and $|bd| = 3q$ for some $q$. Then, the **resultant third string** is given by $ab^2cd^2e = 0^{3l}1^{l+3q}0^{2l} = 0^{2l+q}0^{l-q}1^{l+2q}1^q0^{2l} \notin C$.

**Case 4:** Only 0s from the suffix set of 0s exists in $bcd$ and $|bd| = 3q$ for some $q$. Then, the **resultant third string** is given by, $ab^0cd^0e = 0^{3l}1^l0^{2l-3q} = 0^{2l-q}0^{l+q}1^{l-2q}1^{2q}0^{2l-3q} \notin C$, since $x = 0^{2l-q}$ and $x^R \neq 1^{2q}0^{2l-3q}$.

Comment

**Step 4** of 4

Now, if $i < l+2$, $q = 1$ is taken then, $ab^icd^ie = 0^{3l}1^{l+3(i-1)}0^{2l+(i-1)} = 0^{2l+(i-1)}0^{l-(i-1)}1^{l+2(i-1)} \in C$, Since there are not enough 1s to force then into $x$ and $l+2$ is first time is guaranteed. Hence, there is a **contradiction occurred in each case**. Thus $C$ **is not a context free language**. In other words, it can be said that " $\overline{C}$ **is a context free language**".

## Problem

Recall the CFG $G_4$ that we gave in Example 2.4. For convenience, let's rename its variables with single letters as follows.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T \times F \mid F$$
$$F \rightarrow (E) \mid \mathtt{a}$$

Give parse trees and derivations for each string.

**a.** a

**b.** a+a

**c.** a+a+a

**d.** ((a))

## Step-by-step solution

**Step 1** of 8

Given Grammar $G_4$ is

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T \times F \mid F$$
$$F \rightarrow (E) \mid a$$

**Derivation:** The sequence of substitutions to obtain a string is called a *derivation*.

**Parse Tree:** The pictorial representation of derivation of a string is a *parse tree*.

Comment

**Step 2** of 8

a)

The parse tree to generate string $a$ is as follows:

Comment

**Step 3** of 8

The derivation for the string $a$ is as follows:

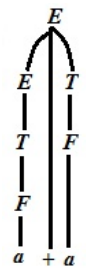
$E \Rightarrow T$
$E \Rightarrow F$
$E \Rightarrow a$

Comment

**Step 4** of 8

b)

The parse tree to generate string $a+a$ is as follows:




Comment

**Step 5** of 8

The derivation for the string $a+a$ is as follows:

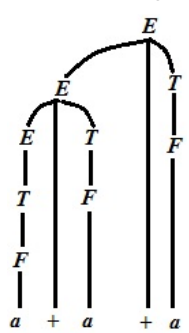
$E \Rightarrow E+T$
$E \Rightarrow T+T$
$E \Rightarrow F+T$
$E \Rightarrow a+T$
$E \Rightarrow a+F$
$E \Rightarrow a+a$

Comment

**Step 6** of 8

c)

The parse tree to generate string $a+a+a$ is as follows:




The derivation for the string $a+a+a$ is as follows:

$E \Rightarrow E+T$
$E \Rightarrow E+T+T$
$E \Rightarrow T+T+T$
$E \Rightarrow F+T+T$
$E \Rightarrow a+T+T$
$E \Rightarrow a+F+T$
$E \Rightarrow a+a+T$
$E \Rightarrow a+a+F$
$E \Rightarrow a+a+a$

Comment

**Step 7** of 8

d)

The parse tree to generate string $((a))$ is as follows:

$$
\begin{array}{c}
E \\ | \\ T \\ | \\ F \\ | \\ (E) \\ | \\ (T) \\ | \\ (F) \\ | \\ ((E)) \\ | \\ ((T)) \\ | \\ ((F)) \\ | \\ ((a))
\end{array}
$$

Comment

**Step 8** of 8

The derivation for the string $((a))$ is as follows:

$$
\begin{aligned}
&E \Rightarrow T \\
&E \Rightarrow F \\
&E \Rightarrow (E) \\
&E \Rightarrow (T) \\
&E \Rightarrow (F) \\
&E \Rightarrow ((E)) \\
&E \Rightarrow ((T)) \\
&E \Rightarrow ((F)) \\
&E \Rightarrow ((a))
\end{aligned}
$$

Comments (2)

## Problem

**a.** Use the languages

$$A = \{\mathtt{a}^m \mathtt{b}^n \mathtt{c}^n \mid m, n \geq 0\} \text{ and } B = \{\mathtt{a}^n \mathtt{b}^n \mathtt{c}^m \mid m, n \geq 0\}$$

together with Example 2.36 to show that the class of context-free languages is not closed under intersection.

**b.** Use part (a) and DeMorgan's law (Theorem 0.20) to show that the class of context-free languages is not closed under complementation.

**THEOREM 0.20**

For any two sets $A$ and $B$, $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

## Step-by-step solution

**Step 1** of 4

**Context Free languages**

a) Given the languages are

$A = \{a^m b^n c^n \mid m, n \geq 0\}$ and

$B = \{a^n b^n c^m \mid m, n \geq 0\}$

Now we will show that both $A$ and $B$ are context-free languages.

In order to show, let us construct grammar that recognizes $A$.

$S \to UT$
$U \to aU \mid \varepsilon$
$T \to bTc \mid \varepsilon$

Observing the above grammar we can say that the language $A$ is a context-free language.

Let us construct grammar that recognizes $B$.

$S \to TU$
$T \to aTb \mid \varepsilon$
$U \to cU \mid \varepsilon$

Observing the above grammar we can say that the language $B$ is a context-free language.

Hence both $A$ and $B$ are context-free languages.

Comment

**Step 2** of 4

Consider $A \cap B = \{a^n b^n c^n \mid n \geq 0\}$.

Now check whether the language $A \cap B$ is a context-free or not using pumping lemma.

Let us assume that $A \cap B$ is a context-free language.

Pumping lemma states that every context-free language has a special value called *pumping length* such that all longer strings in the language can be "pumped",

let $p$ be the pumping length for $A \cap B$.

Consider a string $s = a^p b^p c^p$.

Clearly $s$ is a member of $A \cap B$ and of length at least $p$.

Now we prove that one condition of pumping lemma violated by proving $s$ cannot be pumped.

If we divide $s$ into $wxyz$, condition 2 stipulates that either $v$ or $y$ is non-empty.

Now consider one of the two cases, depending on whether substring $v$ and $y$ contains more than one type of alphabet symbol.

1. If both $v$ and $y$ contain only one type of symbol, $v$ doesn't contain both $a$'s and $b$'s or both $b$'s and $c$'s, and the same holds for $y$. Here the string $uv^2xy^2z$ cannot contain equal number of $a$'s, $b$'s and $c$'s. Therefore it cannot be a member of $A \cap B$ which violates the first condition of the pumping lemma and thus is a contradiction to our hypothesis.

2. If either $v$ or $y$ contain more than one type of symbol $uv^2xy^2z$ may contain equal number of the three alphabet symbols but not in the correct order. Hence it cannot be a member of $A \cap B$ and thus is a contradiction to our hypothesis.

One of the above two case must occur. However, both the cases raised contradiction. This is because of our assumption $A \cap B$ is a context-free language.

Hence our assumption is false and $A \cap B$ is not a context-free language.

Comments (4)

**Step 3** of 4

Hence, we have $A$ and $B$ are context-free languages and $A \cap B$ is not a context-free language. So we can say that the language obtained by intersection of two context-free languages $A$ and $B$ is not a context a context-free language.

**Therefore, the languages $A$ and $B$ are not closed under intersection.**

Comment

**Step 4** of 4

b) Using DeMorgan's law we will show that the languages $A$ and $B$ is not closed under complementation.

DeMorgan's law states that for any two sets $A$ and $B$, $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

We have $A$ and $B$ are two arbitrary context-free languages.

Let these languages are represented in 4-tuple form as $A = (V_1, \Sigma, R_1, S_1)$ and $B = (V_2, \Sigma, R_2, S_2)$ where

- $V_1, V_2$ are finite set of variables of $A$ and $B$ respectively.
- $\Sigma$ is finite set, disjoint from $V_1, V_2$ are terminals of $A$ and $B$ respectively.
- $R_1, R_2$ are finite set of rules of $A$ and $B$ respectively.
- $S_1 \in V_1, S_2 \in V_2$ are the start variables of $A$ and $B$ respectively.

Now construct a grammar $G$ that recognizes $A \cup B$.

So $G = (V, \Sigma, R, S)$ where

- $V = V_1 \cup V_2$
- $R = R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$. Here, $R_1$ and $R_2$ are disjoint.

Now we have to show that $A$ and $B$ are not closed under complementation.

Let us assume that $A$ and $B$ are closed under complementation.

Since, $A$ and $B$ are context-free languages, then $\overline{A}$ and $\overline{B}$ are also context-free-languages. We know that the context-free-languages are closed under union.

So, $\overline{A} \cup \overline{B}$ is closed. Hence $\overline{A} \cup \overline{B}$ is a context-free-language.

Since, $\overline{A} \cup \overline{B}$ is a context-free-language, we have $\overline{\overline{A} \cup \overline{B}}$ is a context-free-language.

Applying DeMorgan's law we get $\overline{\overline{A} \cup \overline{B}} = A \cap B$.

Hence $A \cap B$ is a context-free-language which is a contradiction to part(a).

This contradiction occurred because our assumption is wrong.

Hence $A$ and $B$ are not closed under complementation.

**Therefore, class of context-free-languages is not closed under complementation.**

Comments (2)

**Problem**

Answer each part for the following context-free grammar G.

$$
\begin{aligned}
R &\to XRX \mid S \\
S &\to \mathtt{a}T\mathtt{b} \mid \mathtt{b}T\mathtt{a} \\
T &\to XTX \mid X \mid \varepsilon \\
X &\to \mathtt{a} \mid \mathtt{b}
\end{aligned}
$$

**a.** What are the variables of $G$?

**b.** What are the terminals of $G$?

**c.** Which is the start variable of $G$?

**d.** Give three strings in $L(G)$.

**e.** Give three strings *not* in $L(G)$.

**f.** True or False: $T \Rightarrow \mathtt{aba}$.

**g.** True or False: $T \overset{*}{\Rightarrow} \mathtt{aba}$.

**h.** True or False: $T \Rightarrow T$.

**i.** True or False: $T \overset{*}{\Rightarrow} T$.

**j.** True or False: $XXX \overset{*}{\Rightarrow} \mathtt{aba}$.

**k.** True or False: $X \overset{*}{\Rightarrow} \mathtt{aba}$.

**l.** True or False: $T \overset{*}{\Rightarrow} XX$.

**m.** True or False: $T \overset{*}{\Rightarrow} XXX$.

**n.** True or False: $S \overset{*}{\Rightarrow} \varepsilon$.

**o.** Give a description in English of $L(G)$.

**Step-by-step solution**

**Step 1** of 16

Given the context – free grammar $G$ is

$R \to XRX \mid S$
$S \to aTb \mid bTa$
$T \to XTX \mid X \mid \varepsilon$
$X \to a \mid b$

Comment

**Step 2** of 16

a. The variables of $G$ are $R, S, T$ and $X$

*Variables* are the non-terminal symbols that appear in the rules of the grammar.

Comment

**Step 3** of 16

b. The terminals of $G$ are $a, b$.

*Terminals* are the terminal symbols that appear in the rules of the grammar.

Comments (2)

**Step 4** of 16

c. The start variable of $G$ is $R$.

$R \to XRX \mid S$

Start variable is a variable usually occurs on the left – hand side of the topmost rule.

Comments (3)

**Step 5** of 16

d. Case1: Consider the rule $R \to S$

Substitute S with rule $S \to aTb$

$R \to aTb$

Substitute T with rule $T \to \varepsilon$.

$R \to a \in b$

$R \to ab$

Case2: Consider the rule $R \to S$

Substitute S with rule $S \to bTa$

$R \to bTa$

Substitute T with rule $T \to \varepsilon$.

$R \to b \in a$

$R \to ba$

Case3: Consider the rule $R \to S$

Substitute S with rule $S \to aTb$

$R \to aTb$

Substitute T with rule $T \to X$.

$R \to aXb$

Substitute X with rule $X \to a$.

$R \to aab$

Therefore, the 3 strings in $L(G)$ are $ab$, $ba$ and $aab$.

Comment

**Step 6** of 16

e. The three strings not in $L(G)$ are $aba, b$ and $\varepsilon$. Since these strings cannot be derived from the given grammar *G*.

Comments (6)

**Step 7** of 16

f. False

$T \Rightarrow aba$, the string cannot be derived using $G$.

Comments (1)

**Step 8** of 16

g. True

$T \overset{*}{\Rightarrow} aba$, the string can be derived using $G$.

Comments (16)

**Step 9** of 16

h. False

$T \Rightarrow T$, the string cannot be derived using $G$.

Comment

**Step 10** of 16

i. True

$T \overset{*}{\Rightarrow} T$, the string can be derived using $G$.

Comments (15)

**Step 11** of 16

j. True

$XXX \overset{*}{\Rightarrow} aba$, the string can be derived using $G$.

Comment

**Step 12** of 16

k. False

$X \overset{*}{\Rightarrow} aba$, the string cannot be derived using $G$.

Comments (2)

**Step 13** of 16

l. True

$T \overset{*}{\Rightarrow} XX$, the string can be derived using $G$.

Comments (2)

**Step 14** of 16

m. True

$T \overset{*}{\Rightarrow} XXX$, the string can be derived using $G$.

Comments (1)

**Step 15** of 16

n. False

$S \overset{*}{\Rightarrow} \varepsilon$, the string cannot be derived using $G$.

Comment

**Step 16** of 16

o. $L(G)$ consists of all strings that are not palindromes and are formed over terminal symbols $a$ and $b$.

Comments (3)

## Problem

Give context-free grammars that generate the following languages. In all parts, the alphabet ∑ is {0,1}.

A**a.** {wl w contains at least three 1s}

**b.** {wl w starts and ends with the same symbol}

**c.** {wl the length of w is odd}

A**d.** {wl the length of w is odd and its middle symbol is a 0}

**e.** {wl w = $w^R$, that is, w is a palindrome}

**f.** The empty set

## Step-by-step solution

**Step 1** of 6

The alphabet $\sum$ is given by $\{0, 1\}$

(a)

The context free grammar that generates the language

$\{\omega \mid \omega \text{ contains at least three 1s}\}$ is given by

$$S \to P1P1P1P$$
$$P \to 0P \mid 1P \mid \varepsilon$$

Comments (4)

**Step 2** of 6

(b)

The context free grammar that generates the language $\{\omega \mid \omega \text{ starts and ends with the same symbol}\}$ is given by

$$S \to 0P0 \mid 1P1 \mid 0 \mid 1$$
$$P \to 0P \mid 1P \mid \varepsilon$$

Comments (3)

**Step 3** of 6

(c)

The context free grammar that generates the language

$\{\omega \mid \text{the length of } \omega \text{ is odd}\}$ is given by

$$S \to 0 \mid 1 \mid 00S \mid 01S \mid 10S \mid 11S$$
$$(\text{or})$$
$$S \to 0 \mid 1 \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$$

Comments (14)

**Step 4** of 6

(d)

The context free grammar that generates the language $\{\omega \mid \text{the length of } \omega \text{ is odd and its middle symbol is a } 0\}$ is given by

$S \to 0 \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$

Comments (5)

**Step 5** of 6

(e)

The context free grammar that generates the language $\{\omega \mid \omega = \omega^R, \text{that is, } \omega \text{ is a palindrome}\}$ is given by

$S \to 0 \mid 1 \mid 0S0 \mid 1S1 \mid \varepsilon$

Comments (12)

**Step 6** of 6

(f)

The context free grammar that generates the language

$\{\ \}$ is given by

$S \to S$

Comments (2)

## Problem

Give informal descriptions and state diagrams of pushdown automata for the languages in Exercise 2.4.

## Step-by-step solution

**Step 1** of 10

**Given:**

$L = \{w \mid w\}$, which contains minimum three 1's

Context free grammar of the above language is as shown:

$S \to R1R1R1R$
$R \to 0R \mid 1R \mid \in$

**Informal Description:**

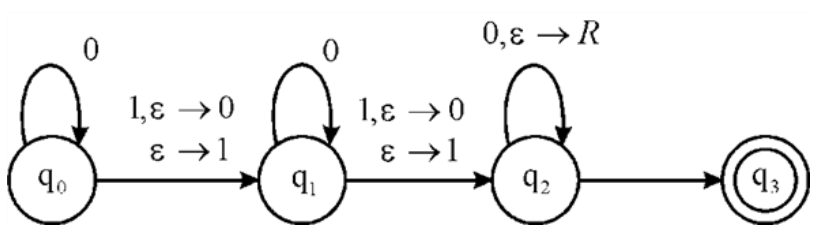
In the above grammar there is no need of stack. So, in the above grammar only the input is being read and that input is accepted only when there is an atleast three 1's. If atleast three 1's are present in any string then push down automata will accept that grammar otherwise it will reject.

**Pushdown Automata:**

Pushdown automata of the informal description is as shown:

Comment

**Step 2** of 10




Comments (6)

**Step 3** of 10

In the above pushdown automata, the state $q_0$ is the initial state and $q_3$ is the final state. Transition is done from $q_0$ to $q_3$ and string is accepted by the push down automata if there is three 1's otherwise string will not be accepted.

Comment

**Step 4** of 10

**Given:**

$L = \{w \mid w\}$, whose starting and ending symbol are same

Context free grammar of the above language is as shown:

$S \to 0A0 \mid 1A1 \mid 0 \mid 1$

$A \to 0A \mid 1A \mid \in$

**Informal Description:**

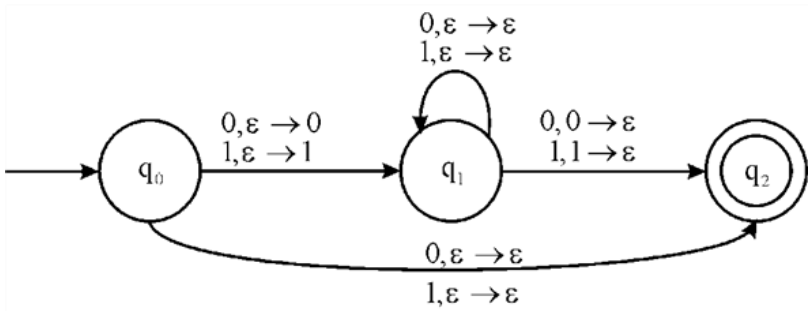
• In the above grammar user made an assumption that there is only one symbol in the string with can be accepted without the use of the stack.

• If there is more than one symbol in the string then all the symbol are sequentially put into the stack.

• Every time, the user identifies whether it is the last symbol which is to be read. Suppose it is the last symbol which is to be read and this symbol matches with the symbol which is present on the stack and if all the input has been retrieved and no input is being left then the user accepts the input otherwise pushdown automata will reject that input.

**Pushdown Automata:**

Pushdown automata of the informal description is as shown:




In the above push down automata, the state $q_0$ is the starting state and $q_2$ is the final state. In the first step user push the element into the push down automata after user pop the symbol from the stack and check whether it match with the final state or not. If it matches with the final stage then string is accepted.

Comments (6)

**Step 5** of 10

**Given:** $q_0$

Context free grammar of the above language is as shown:

$S \to 0A \mid 1A$

$A \to 0S \mid 1S \mid \in$

**Informal Description:**

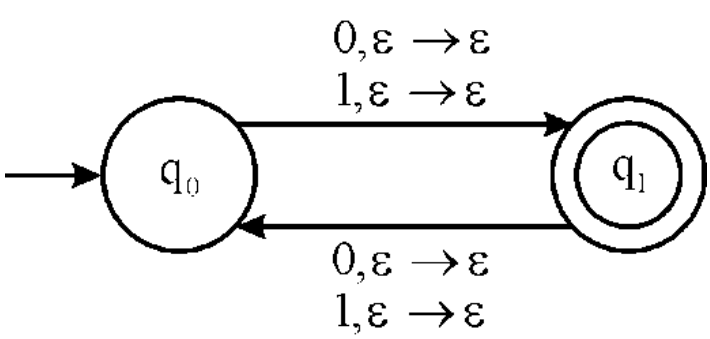
In the above grammar there is no need of stack. So, in the above grammar only the input is being read and that input is accepted only when there is an odd length. If the first symbol is being read or other symbol is being read then it is the final symbol read.

**Pushdown Automata:**

Pushdown automata of the informal description is as shown:

Comment

**Step 6** of 10




In the above pushdown automata, the state $q_0$ is the initial stage and $q_1$ is the final stage.

Comment

**Given:**

$L = \{w \mid \text{the length of } w \text{ is odd and its middle symbol is } 0\}$

Context free grammar of the above language is as shown:

$S \to 0S0 \mid 0S1 \mid 1S0 \mid 1S1 \mid 0$

**Informal Description:**

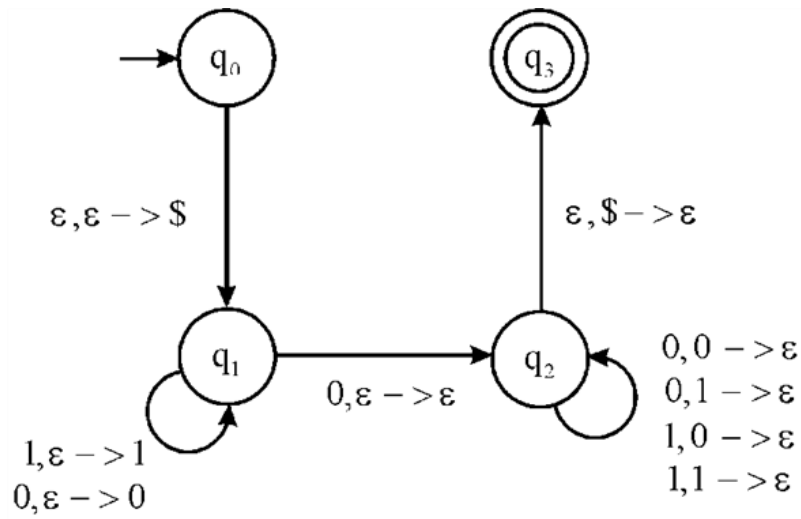
• In the above grammar input is scanned by the push down automata and symbol is placed into the stack. There is some point in the push down automata where it guesses the input which is placed in the middle.

• It checks whether the symbol which is present in the middle contains 0 or not. If 0 is found as a middle element then all the string are scan sequentially and every time character is pop from the stack which has been scanned.

• If the stack after poping is empty and there is no input which is to be read by the stack then it checks whether 0 is the middle symbol in the string. If it is then the string is accepted otherwise the pushdown automata will reject the string.

**Pushdown Automata:**

Pushdown automata of the informal description is as shown:




In the above push down automata, the state $q_0$ is the initial state and $q_3$ is the final state. In each step it checks whether the middle symbol is 0. If the stack is empty and middle symbol is 0 then the string is accepted by the push down automata otherwise pushdown automata will reject the input.

Comments (1)



**Given:**

$L = \{w \mid w = w^R\}$, and $w$ is the palindrome.

**Informal Description:**

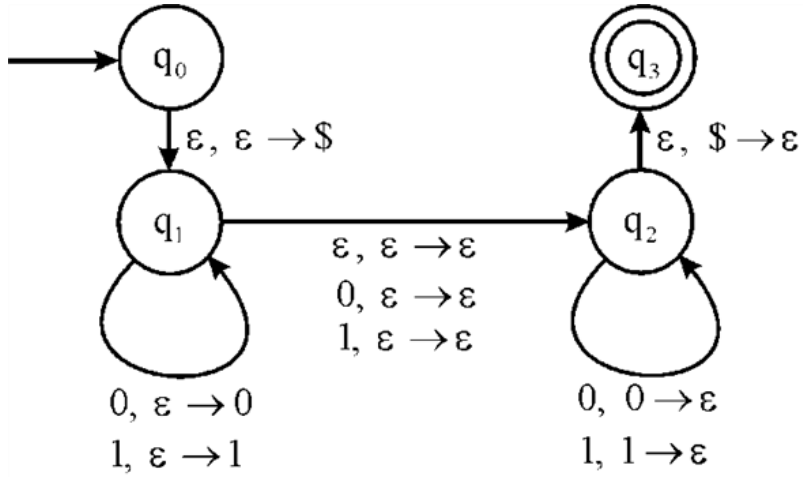
• In the above grammar first half of the palindrome is pushed into the stack and after that underministically middle is being guessed.After that poping of the element is done from the second half of the palindrone.

• The first half of the input and the second half of the input should match with each other.

• If it matches then stack is made empty and after that push down automata accepts that particular string. If it does not matches then pushdown automata then reject that particular string.

**Pushdown Automata:**

Pushdown automata of the informal description is as shown:




• In the above push down automata $q_0$ is the starting stage and $q_3$ is the ending stage. In the stage $q_1$ first half is pushed into the stack excluding the middle symbol if the palindrome is odd.

• If the string is not odd then user guesses where the middle point of the input is non-deterministically.

• After determining the middle part of the input user goes to state $q_2$ without the consumption of any input if the string is found to be even.

• In the state $q_2$ popping of the symbol is done from the stack and it is being checked whether it matches with the input state in the input in the beginning.

•

Comments (2)

**Step 9** of 10

If the first half and second half of the input is matched, then the palindrome is accepted by the push down automata otherwise palindrome is rejected by the push down automata.

Comment

**Step 10** of 10

**Given:**

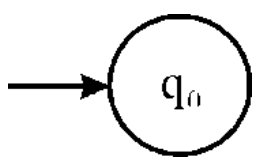
*L* is an empty set which implies $S \to S$

**Informal Description:**

In the above grammar as the language accept only an empty set so it implies it does not terminate any derivation. If non of the derivation is terminated then no any string is accepted by the CFG which also includes an empty string.

**Pushdown Automata:**

Pushdown automata of the informal description is as shown:




Comments (1)

## Problem

Give context-free grammars generating the following languages.

A **a.** The set of strings over the alphabet {a,b} with more a's than b's

**b.** The complement of the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$

A **c.** $\{w\#x \mid w^{\mathcal{R}} \text{ is a substring of } x \text{ for } w, x \in \{0,1\}^*\}$

**d.** $\{x_1\#x_2\#\cdots\#x_k \mid k \geq 1, \text{ each } x_i \in \{\mathtt{a},\mathtt{b}\}^*, \text{ and for some } i \text{ and } j,\ x_i = x_j^{\mathcal{R}}\}$

## Step-by-step solution

**Step 1** of 2

Given language is

"The set of strings over the alphabet $\{a,b\}$ with more $a$'s than $b$'s"

The context – free grammar generating the given language is

$S \to Aa \mid BS \mid SBA$
$A \to Aa \mid \in$
$B \to \in \mid BB \mid bBa \mid aBb$

In the above grammar $S$ will generate all strings with as many $a$'s as $b$'s. $R$ Forces an extra $a$ which gives the required strings of the language.

Given language is

"The compliment of the language $\{a^n b^n : n \geq 0\}$"

Let $L$ be the language that is a compliment of given language. $L$ can be obtained as $L = \{a^n b^m : n \neq m\} \cup \{(a \cup b)^* ba (a \cup b)^*\}$

Let us conside

$L_1 = \{a^n b^m : n \neq m\}$

$L_2 = \{(a \cup b)^* ba (a \cup b)^*\}$

The context – free grammar generating the language $L_1$ is

$S_1 \to aS_1b \mid T \mid U$
$T \to aT \mid a$
$U \to Ub \mid a$

The context – free grammar generating the language $L_2$ is

$S_2 \to RbaR$
$R \to RR \mid a \mid b \mid \varepsilon$

Therefore, the required context – free grammar generating the language $L$ is given by

$L = L_1 \cup L_2$

$S \to S_1 \mid S_2$
$S_1 \to aS_1b \mid T \mid U$
$S_2 \to RbaR$
$T \to aT \mid a$
$U \to Ub \mid a$
$R \to RR \mid a \mid b \mid \varepsilon$

Given language is

$\{w\#x : w^R \text{ is a substring of } x \text{ for } w, x \in \{0,1\}^*\}$

The context – free grammar generating the given language is

$R \to SX$
$S \to 0S0 \mid 1S1 \mid \# X$
$X \to XX \mid 1 \mid 0 \mid \varepsilon$

The nonterminal $S$ ends only with $\# X$, $S$ must generate a string whose beginning and end are mirror images. Since $X$ generates $(0 \cup 1)^*$, the symbol $S$ generates all strings of the form $w\#(0 \cup 1)^* w^R$. The above grammar generates all the substrings of $x$ for $w, x \in \{0,1\}^*$.

Comments (4)

**Step 2** of 2

Given language is

$$\left\{x_1 \# x_2 \# \cdots \# x_k \mid k \geq 1, \text{ each } x_i \in \{a,b\}^*, \text{ and for some } i \text{ and } j, x_i = x_j^R\right\}$$

The context – free grammar generating the given language is

$R \to S \mid J\#S\#J \mid J\#S \mid S\#J$
$S \to aSa \mid bSb \mid \# \mid \#J\#$
$J \to aJ \mid bJ \mid \#J \mid \varepsilon$

The strings in the language contain matching pair of strings with at least one # between them. Before, after and between the matching pairs there can be any number of strings of $a$'s and $b$'s separated by #. Because the strings can be of any length, the strech of the strings of $a$'s, $b$'s and #'s can be of any length. The symbol $S$ generates a matching pair, with strings of $a$'s, $b$'s and #'s optionally inserted in the middle. The symbol $J$ generates strings of $a$'s, $b$'s and #'s. The string generated by $J$ may start or end with $a$ or $b$, so rules for $M$ and $S$ must ensure that the symbol $J$ is always separated properly from the two matching strings.

Comments (1)

## Problem

Give informal English descriptions of PDAs for the languages in Exercise 2.6.

## Step-by-step solution

**Step 1** of 4

**a.**

**Consider the following language:**

"The set of strings over the alphabet $\{a,b\}$ with more $a$'s than $b$'s"

This is the language where the number of $a$'s is greater then the number of $b$'s. Construct the PDA (Push Down Automata), that accepts the above language. It uses its stack to count the number of $a$'s minus the number of $b$'s. It enters an accepting state whenever this count is positive or top of the stack is $a$.

**In more detail, it operates as follows:**

• The PDA scans across the input.

• If it sees a $a$ and its top of the stack symbol is either $a$ or empty than, push $a$ into the stack.

• If it scans a $a$ and its top stack symbol is a $b$, it pops the stack.

• If it sees a $b$ and its top of the stack symbol is either $b$ or empty than, push $b$ into the stack.

• If it sees a $b$ and its top stack symbol is $a$, it pops the stack.

After the PDA finishes the input, if $a$ is on top of the stack, it accepts. Otherwise, it rejects.

Therefore, given language is recognized by the above PDA.

Comment

**Step 2** of 4

**b.**

**Consider the following language:**

"The compliment of the language $\{a^n b^n : n \geq 0\}$"

First, we will construct the compliment of the given language. That is number of $a$'s is not equal to the number of $b$'s. The compliment of this language can obtain by breaking the given language into the union of several simpler languages.

The PDA uses its stack to count the number of $a$'s and the number of $b$'s. It enters an accepting state whenever this count is not equal.

**In more detail, it operates as follows:**

• The PDA scans across the input.

• If it sees a $a$ and its top of the stack symbol is either $a$ or empty than, push $a$ into the stack.

• If it sees a $b$ and its top stack symbol is $a$, it pops the stack.

• If it sees a $b$ and its top of the stack symbol is either $b$ or empty than, push $b$ into the stack.

After the PDA finishes the input, if $a$ or $b$ is on top of the stack, it accepts. Otherwise, it rejects.

Therefore, if the end of the input is reached and stack is not empty than, it accepts. Otherwise it rejects because number of $a$'s and the number of $b$'s in the input are equal.

Comments (2)

**Step 3** of 4

**c.**

**Consider the following language:**

$\left\{ w\#x : w^R \text{ is a substring of } x \text{ for } w,x \in \{0,1\}^* \right\}$

The language is obtained by constructing the strings $w^R$ which is a substring $x$. The PDA scans across the input string and finds the input is a substring of $x$.

In more detail, it operates as follows:

• The PDA scans across the input string and pushes every symbol it reads until it reads until it reads a $\#$.

• If $\#$ is never encountered, it rejects.

• Then the PDA skips over part of the input, nondeterministically deciding when to stop skipping.

• At that point, it compares the next input finishes while the stack is nonempty, this branch of the computations rejects.

After the PDA finishes the input, if the stack becomes empty, the machine reads the rest of the input and accepts.

Therefore, given language is recognized by the above PDA.

Comment

**Step 4** of 4

d.

**Consider the following language:**

$$\left\{x_1 \# x_2 \# \cdots \# x_k \mid k \geq 1, \text{ each } x_i \in \{a,b\}^*, \text{ and for some } i \text{ and } j, x_i = x_j^R\right\}$$

The language can obtain by constructing matching pair of strings with at least one $\#$ between them. Before, after and between the matching pairs there can be any number of strings of $a$'s and $b$'s separated by $\#$. The PDA scans across the input string and finds the input is with at least one $\#$ is between them.

**In more detail, it operates as follows:**

• The PDA scans across the input string and pushes every symbol it reads until it reads until it reads a $\#$.

• If # is never encountered, it rejects.

• If an input $\#$ is read, the nondeterministically skips to another # in the input and continue reading input symbols, now comparing them with symbols popped off the stack. If all agree and stack becomes empty at the same time as either end of the input of # symbol is reached, accept, otherwise reject.

After the PDA finishes the input, if the stack becomes empty, the machine reads the rest of the input and accepts.

Therefore, given language is recognized by the above PDA.

Comment

## Problem

Show that the string the girl touches the boy with the flower has two different leftmost derivations in grammar $G_2$ on page 103. Describe in English the two different meanings of this sentence.

## Step-by-step solution

**Step 1** of 2

Leftmost derivative is use for replacing the leftmost variable. Each and every variable is replaced on the consecutive step.

**Given:** the girl touches the boy with flower

**Proof:** The above given string has two leftmost derivative

Grammars describing the fragment of English language are as follows:

$$\begin{aligned}\langle\text{SENTENCE}\rangle &\to \langle\text{NOUN-PHRASE}\rangle\langle\text{VERB-PHRASE}\rangle\\ &\to \langle\text{CMPLX-NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\\ &\to \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\\ &\to \text{the}\langle\text{NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\end{aligned}$$

$$\begin{aligned}&\to \text{the girl}\langle\text{VERB-PHRASE}\rangle\\ &\to \text{the girl}\langle\text{CMPLX-VERB}\rangle\\ &\to \text{the girl}\langle\text{VERB}\rangle\langle\text{NOUN-PHRASE}\rangle\\ &\to \text{the girl touches}\langle\text{NOUN-PHRASE}\rangle\\ &\to \text{the girl touches}\langle\text{CMPLX-NOUN}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches}\langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches the }\langle\text{NOUN}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches the boy}\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches the boy}\langle\text{PREP}\rangle\langle\text{CMPLX-NOUN}\rangle\\ &\to \text{the girl touches the boy with}\langle\text{CMPLX-NOUN}\rangle\\ &\to \text{the girl touches the boy with }\langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\\ &\to \text{the girl touches the boy with the}\langle\text{NOUN}\rangle\\ &\to \text{the girl touches the boy with the flower}\end{aligned}$$

The above phrase is the first derivative of the leftmost derivation.

Comment

**Step 2** of 2

Grammars describing the fragment of English language are as follows:

$$\begin{aligned}\langle\text{SENTENCE}\rangle &\to \langle\text{NOUN-PHRASE}\rangle\langle\text{VERB-PHRASE}\rangle\\ &\to \langle\text{CMPLX-NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\\ &\to \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\\ &\to \text{the}\langle\text{NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\end{aligned}$$

$$\begin{aligned}&\to \text{the girl}\langle\text{VERB-PHRASE}\rangle\\ &\to \text{the girl}\langle\text{CMPLX-VERB}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl}\langle\text{VERB}\rangle\langle\text{NOUN-PHRASE}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches}\langle\text{NOUN-PHRASE}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches}\langle\text{CMPLX-NOUN}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches}\langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches the}\langle\text{NOUN}\rangle\langle\text{PREP-PHRASE}\rangle\\ &\to \text{the girl touches the boy}\langle\text{PREP-PHRASE}\rangle\end{aligned}$$

$\rightarrow$ the girl touches the boy $\langle$PREP$\rangle\langle$CMPLX-NOUN$\rangle$

$\rightarrow$ the girl touches the boy with $\langle$CMPLX-NOUN$\rangle$

$\rightarrow$ the girl touches the boy with $\langle$ARTICLE$\rangle\langle$NOUN$\rangle$

$\rightarrow$ the girl touches the boy with the $\langle$NOUN$\rangle$

$\rightarrow$ the girl touches the boy with the flower

The above phrase is the second derivative of the leftmost derivation.

Comments (1)

## Problem

Give a context-free grammar that generates the language

$$A = \{\mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \mid i = j \text{ or } j = k \text{ where } i, j, k \geq 0\}.$$

Is your grammar ambiguous? Why or why not?

## Step-by-step solution

**Step 1** of 1

The language given in the problem is as follows:

$A = \left\{a^i b^j c^k \mid \text{ i=j } or \text{ j} = k \text{ for i} \geq 0, \text{j} \geq 0,\ k \geq 0\right\}$

**The language *A* can be split into two languages which are defined as follows**:

$A_1 = \left\{a^i b^j c^k \mid i, j, k \geq 0, i = j\right\}$

and

$A_2 = \left\{a^i b^j c^k \mid i, j, k \geq 0, j = k\right\}$

Using the language *A1* and *A2* the user can construct a CFG for $A_1$ and $A_2$.

The grammar for language *A* is the union of grammar of two languages which is defined as follows:

$S \to S_1 \mid S_2$

In the language $A_1$ the values of i and j are equal so there must be equal number of a's and b's in the language $A_1$.

**CGF for the language *A* $_1$ is as follows:**

$S_1 \to S_1 c \mid E \mid \in$
$E \to aEb \mid \in$

Similarly, in the language $A_2$ the values of j and k are equal so there must be equal number of b's and c's in the language $A_2$.

**CGF for the language *A* $_2$ is as follows:**

$S_2 \to aS_2 \mid F \mid \in$
$F \to bFc \mid \in$

Since for generating a string $w = a^n b^n c^n$ using the language *A*, either $S_1$ or $S_2$ can be used.

Therefore, the context free grammar for the language *A* is ambiguous.

Comment

## Problem

Give an informal description of a pushdown automaton that recognizes the language A in Exercise 2.9.

## Step-by-step solution

**Step 1** of 1

2287-2-10E AID: 824

RID: 944

Given language is

$A = \{a^i b^j c^k \mid i = j \text{ or } j = k \text{ where } i, j, k \geq 0\}$

The language is union of two languages $\{a^i b^i c^k \mid i, k \geq 0\}$ and $\{a^i b^k c^k \mid i, k \geq 0\}$.

Let $A_1 = \{a^i b^i c^k \mid i, k \geq 0\}$ and $A_2 = \{a^i b^k c^k \mid i, k \geq 0\}$.

The informal description of the PDA that recognizes the language $A_1$.

In more detail, it operates as follows:

• Read and push $a$'s.

• Read $b$'s, while popping $a$'s.

• If $b$'s finish when stack is empty, skip $c$'s on input and accept.

The informal description of the PDA that recognizes the language $A_2$.

In more detail, it operates as follows:

• Skip $a$'s on input.

• Read and push $b$'s.

• Read $c$'s, while popping $b$'s.

• If $c$'s finish when stack is empty, accept.

The informal description of the PDA that recognizes the language $A$ is the combinition of both the languages $A_1$ and $A_2$.

In more detail, it operates as follows:

1. Nondeterministically branck to either step 2 or step 6.

2. Read and push $a$'s.

3. Read $b$'s, while popping $a$'s.

4. If $b$'s finish when stack is empty, skip $c$'s on input and accept.

5. Skip $a$'s on input.

6. Read and push $b$'s.

7. Read $c$'s, while popping $b$'s.

8. If $c$'s finish when stack is empty, accept.

Comment

## Problem

Convert the CFG G4 given in Exercise 2.1 to an equivalent PDA, using the procedure given in Theorem 2.20.

THEOREM **2.20**

A language is context free if and only if some pushdown automaton recognizes it.

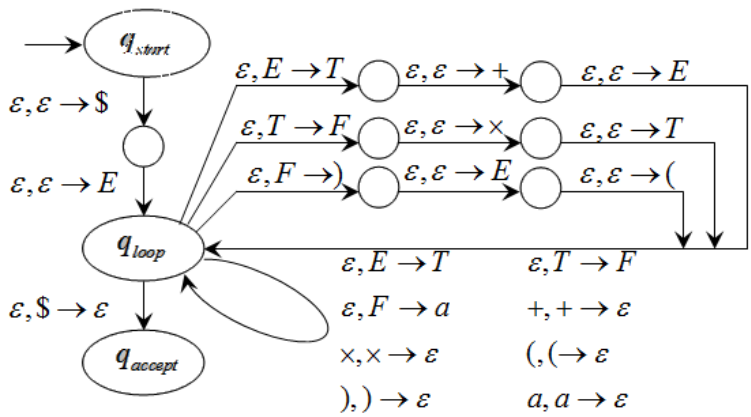
## Step-by-step solution

**Step 1** of 2

Given CFG (Context-free grammar) $G_4$ is

$E \to E+T \mid T$
$T \to T \times F \mid F$
$F \to (E) \mid a$

**Equivalent PDA for the CFG $G_4$ is as follows:**




Comments (5)

**Step 2** of 2

**Explanation:**

1. A shorthand notation is used for pushing multiple symbols onto the stack.

2. Initially, at the start variable on the stack a marker symbol '$' is inserted. The start state is $q_{start}$. The transition function is $\delta(q_{start}, \in, \in) = \{(q_{loop}, S\$)\}$

3. If the stack top is a non-terminal variable E. Select one of the rules of E and substitutes its value on the right hand side of the rule. Repeat this process until the end of the string.

The transition function is $\delta(q_{loop}, \in, E) = \{(q_{loop}, w) \mid E \to w \text{ is a rule in CFG}\}$

**Example:**

• Consider the rule $E \to E+T$.

• Another rule for *E* is $E \to T$. Substitute the value of *E* in the above rule.

• Then, the equation becomes $E \to T+T$.

4. If the stack top is a terminal variable such as (,),a,+ and x the next symbol is read from the input rule. Repeat step-3 if again a non-terminal variable is encountered.

The transition function is $\delta(q_{loop}, a, a) = \{(q_{loop}, \in)\}$.

5. If the stack top is a '$' symbol, the accept state is entered because, the input is read completely.

The transition function is $\delta(q_{loop}, \in, \$) = \{(q_{accept}, \in)\}$.

Comment

## Problem

Convert the CFG G given in Exercise 2.3 to an equivalent PDA, using the procedure given in Theorem 2.20.

THEOREM **2.20**

A language is context free if and only if some pushdown automaton recognizes it.

## Step-by-step solution

**Step 1** of 2

**CFG** stands for context free grammar. It is a set of recursive rules which are used to create a string pattern.

• A CFG contains set of terminals and no-terminals.

• Generally, Non terminals or variables are represented with capital alphabets whereas terminals are represented with small alphabets.

• The languages which use context free grammar are called as context free language.

• Machines which recognize the context free language are called as **push down automata**. It is used to provide additional power to CFG.

Comment

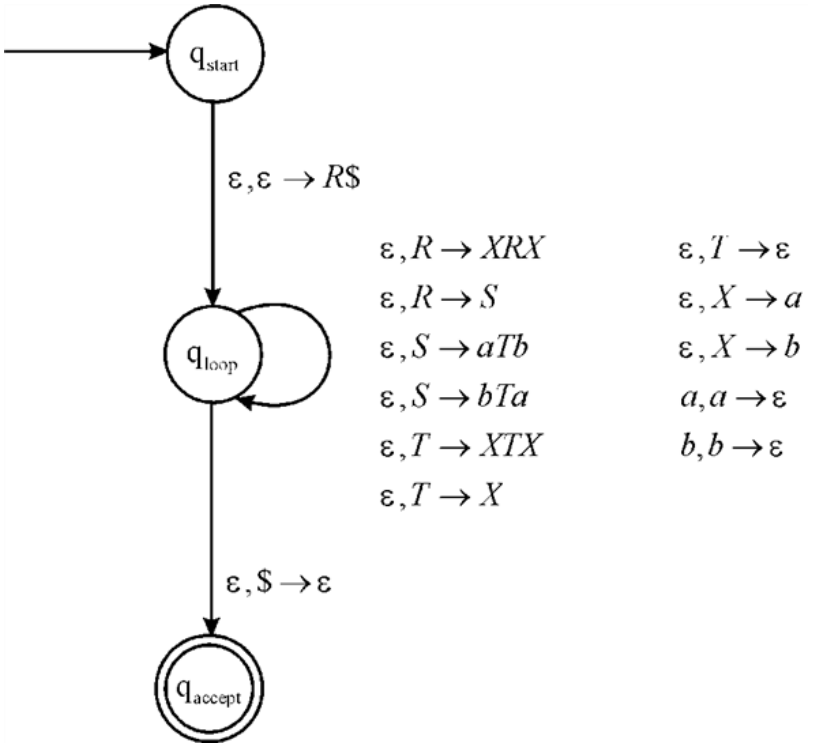
**Step 2** of 2

**Conversion of CFG to PDA:**

Consider the following context free grammar:

$$R \to XRX \mid S$$
$$S \to aTb \mid bTa$$
$$T \to XTX \mid X \mid \varepsilon$$
$$X \to a \mid b$$

Consider the following diagram to represent the equivalent PDA of the above CFG:




• The above transition rule allows someone to reduce the grammar that is to replace non-terminals or variables to the right-hand side terminals.

• Transition for the terminal symbols such as $(a,b)$ permits someone for matching the input symbol to the terminal symbol.

• A path of PDA of string *w* can only be accepted if an input string *w* can only be generated by grammar *G*.

## Problem

Let G = (V,∑,R, S) be the following grammar. V = {S, T, U}; ∑ = {0, #}; and R is the set of rules:

$$S \to TT \mid U$$
$$T \to 0T \mid T0 \mid \#$$
$$U \to 0U00 \mid \#$$

**a.** Describe L(G) in English.

**b.** Prove that L(G) is not regular.

## Step-by-step solution

**Step 1** of 4

Let $G=(V,\Sigma,R,S)$ be the grammar. $V=\{S,T,U\}$; $\Sigma=\{0,\#\}$. $R$ is the set of rules.

$S \to TT \mid U$
$T \to 0T \mid T0 \mid \#$
$U \to 0U00 \mid \#$

Comment

**Step 2** of 4

Description of $L(G)$ in English is given as

$L(G)$ is the set of all strings containing $0\text{'s and } \#\text{'s}$ of the form(s)

(a) Strings containing exactly two $\#$'s and any number of $0\text{'s}$.

(b) Strings containing exactly one $\#$ and the number of $0\text{'s}$ on the left of $\#$ are half the number of $0\text{'s}$ to the right.

Comments (1)

**Step 3** of 4

To prove the language $L(G)$ is not regular, consider by contradiction $L(G)$ is regular.

Let $A=L(G)\cap 0^*\#0^*$.

From our consideration $L(G)$ is regular. So $A$ is regular.

Since $A$ is regular, using pumping lemma let $p$ be the pumping length of the regular language.

Consider the sting $w=0^p\#0^{2p}$.

Clearly length of the string $w\in A$ is greater the $p$. That is $|w|>p$.

So, by pumping lemma we have $w=xyz$ such that

$|xy|\le p,\ y\ne\varepsilon \text{ and } xy^iz\in L(G) \text{ for all } i\ge 0.$

Comment

**Step 4** of 4

Since $A$ is a regular language, consider the possible ways of cutting the string $w$.

• If $x$ contains the character $\#$, then $y$ will be on the right side of $\#$. Pumping $y$ down, increases the number of $0's$ on the left such that number of $0's$ on the left of $\#$ are not equal to half the number of $0's$ to the right.

• If $y$ contains the character $\#$, then pumping $y$ down, increases the number of $\#$'s in the string.

• If $z$ contains the character $\#$, then $y$ will be on the left side of $\#$. Pumping $y$ down, decreases the number of $0's$ on the right such that number of $0's$ on the left of $\#$ are not equal to half the number of $0's$ to the right.

The resulting string of all the above cases does not belong to $A$. So, we can say that $A$ does not satisfy pumping lemma. Hence $A$ is not a regular language.

This is a contradiction to our statement that $L(G)$ regular language.

**Therefore, $L(G)$ is not a regular language.**

Comment

## Problem

Convert the following CFG into an equivalent CFG in Chomsky normal form, using the procedure given in Theorem 2.9.

$$A \rightarrow BAB \mid B \mid \varepsilon$$
$$B \rightarrow 00 \mid \varepsilon$$

THEOREM **2.9**

Any context-free language is generated by a context-free grammar in Chomsky normal form.

## Step-by-step solution

**Step 1** of 7

Given CFG (Context-free Grammar) is

$A \rightarrow BAB \mid B \mid \varepsilon$
$B \rightarrow 00 \mid \varepsilon$

Now, construct an equivalent CFG (Context-free Grammar) in Chomsky normal form.

**Chomsky normal form:**

A context i– free grammar s in **Chomsky normal form** if every rule is of the form

$A \rightarrow BC$
$A \rightarrow a$

Here, $a$ is terminal,

$A, B$ and $C$ are variables,

In addition, it permits the rule $S \rightarrow \varepsilon$, here $S$ is the start variable.

convert the given CFG into an equivalent CFG in Chomsky normal form.

Comment

**Step 2** of 7

Let's add a new start variable $S_0$ and the rule $S_0 \rightarrow A$.

Thus the obtained grammar is

$S_0 \rightarrow A$
$A \rightarrow BAB \mid B \mid \varepsilon$
$B \rightarrow 00 \mid \varepsilon$

In the addition of new start variable guarantees that the start variable doesn't occur on the right-hand side of a rule.

Comment

**Step 3** of 7

Removing all rules that containing $\varepsilon$.

Removing $A \rightarrow \varepsilon$ and $B \rightarrow \varepsilon$ gives

$S_0 \rightarrow A \mid \varepsilon$
$A \rightarrow BAB \mid BA \mid AB \mid A \mid B \mid BB$
$B \rightarrow 00$

The rule $S_0 \rightarrow \varepsilon$ is accepted since $S_0$ is the start variable and that is allowed in Chomsky normal form.

Comments (1)

**Step 4** of 7

Now remove the unit rules.

Removing $A \to A$ gives

$$S_0 \to A \mid \varepsilon$$
$$A \to BAB \mid BA \mid AB \mid B \mid BB$$
$$B \to 00$$

Removing $S \to B$ gives

$$S_0 \to A \mid \varepsilon$$
$$A \to BAB \mid BA \mid AB \mid 00 \mid BB$$
$$B \to 00$$

Removing $S_0 \to S$ gives

$$S_0 \to BAB \mid BA \mid AB \mid 00 \mid BB \mid \varepsilon$$
$$A \to BAB \mid BA \mid AB \mid 00 \mid BB$$
$$B \to 00$$

Comments (2)

**Step 5** of 7

Now replace ill placed terminals 0 by variable $U$ with new

$$S_0 \to BAB \mid BA \mid AB \mid UU \mid BB \mid \varepsilon$$
$$A \to BAB \mid BA \mid AB \mid UU \mid BB$$
$$B \to UU$$
$$U \to 0$$

Comment

**Step 6** of 7

Shorten the right-hand side of rules with only 2 variables each.

To shorten the rules, replace $S_0 \to BAB$ with two rules $S_0 \to BA_1$ and $A_1 \to AB$.

The rule $A \to BAB$ is replaced by the two rules $A \to BA_2$ and $A_2 \to AB$.

After replacing these rules, the final Context-free grammar in Chomsky normal form is $G = (V, \Sigma, R, S_0)$,

Here the set of variables is $V = \{S_0, S, B, U, A_1, A_2\}$,

the start variable is $S_0$.

The set of terminals is $\Sigma = \{0\}$, and the rules $R$ are given by

Comment

**Step 7** of 7

$$S_0 \to BA_1 \mid BA \mid SB \mid UU \mid BB \mid \varepsilon$$
$$A \to BA_1 \mid BA \mid SB \mid UU \mid BB$$
$$B \to UU$$
$$U \to 0$$
$$A_1 \to AB$$

This is the final CFG in Chomsky normal form equivalent to the given CFG.

Comments (4)

## Problem

Give a counterexample to show that the following construction fails to prove that the class of context-free languages is closed under star. Let A be a CFL that is generated by the CFG G = (V,∑,R, S). Add the new rule S → SS and call the resulting grammar G′. This grammar is supposed to generate $A^*$.

## Step-by-step solution

**Step 1** of 2

**Example to show the failure of closure of Context free language under Star**

Context free language is the language which is generated by CFG or the context free grammar. It is possible to get different types of context free languages from different types of context free grammar.

**Given:** Here, it is given that *A* is a context free language and the context free grammar which is responsible for its generation is given as $G=(V,\Sigma,R,S)$ .

Here it is required to show the failure of the construction in proving the closure of the context free language is closed under star.

Comment

**Step 2** of 2

**Proof:**

This can be done by adding a new rule as $S \to SS$ and the grammar which will be obtained as resultant will be called as $G'$. This grammar is expected to generate $A^*$.

Now, the counter example can be considered by supposing the CFL as A and the grammar corresponding to it along with its production rules as $G=\{\{S\},\{(,)\},\{S \Rightarrow (S), S \Rightarrow \in\},S\}$.

Now, as required and given add the new production rule as $S \to SS$. As the new rule is being added a new grammar represented as $G'$ will be obtained.

**This grammar will generate the language containing the string symbols as $((\ ),(\ ))$ . And, thus, this grammar is not capable of generating the star closure of *A* represented by $A^*$ .**

**Conclusion:** Thus, it can be concluded that the new grammar $G'$ fails to prove that the class of context free languages is closed under star.

**This condition is true only if the $A^*$ contains empty string else false. So, $A^*$ must contains the empty string.**

Comment

## Problem

Show that the class of context-free languages is closed under the regular operations, union, concatenation, and star.

## Step-by-step solution

**Step 1** of 3

In order to show that context free language is closed under union operation considers two starts variable $S_1$ and $S_2$ for the two different languages $L_1$ and $L_2$.

Grammar for union operation is as shown:

$$S \to S_1 \mid S_2$$

If both the language belongs to the context free language then union of both the language should belong to context free language.

By the above definition if user generates $S_1$ and $S_2$ string or both then in that case union of both the language is generated.

Hence, $L_1 \cup L_2 \in CFL$.

**This implies context free language is closed under union operation.**

Comments (1)

**Step 2** of 3

In order to show that context free language is closed under concatenation operation considers two starts variable $S_1$ and $S_2$ for the two different languages $L_1$ and $L_2$.

Grammar for union operation is as shown:

$$S \to S_1 S_2$$

If both the language belongs to the context free language then concatenation of both the language should belong to context free language.

$$\forall L_1, L_2 \in CFL$$

$$\{w_1 w_2 : w_1 \in L_1 \wedge w_2 \in L_2\} \in CFL$$

By the above definition if user generates $S_1$ string for language $L_1$ followed by $S_2$ string of language $L_2$. Then it concatenation of both the language is generated.

Hence, $\{w_1 w_2 : w_1 \in L_1 \wedge w_2 \in L_2\} \in CFL$

**This implies context free language is closed under concatenation operation.**

Comments (1)

**Step 3** of 3

In order to show that context free language is closed under star operation.

Consider one start variable $S_1$ for the languages $L_1$

Grammar for union operation is as shown:

$$S \to S_1 S \mid \in$$

If the language belongs to the context free language then star of the language should belong to context free language.

$$\forall L_1 \in CFL$$

By the above definition, if user generates zero or many string which is the definition of the star.

**This implies that context free language is closed under star operation.**

Comments (1)

## Problem

Use the results of Exercise 2.16 to give another proof that every regular language is context free, by showing how to convert a regular expression directly to an equivalent context-free grammar.

## Step-by-step solution

**Step 1** of 3

Consider the following data:

From Exercise 2.16, it is clear that Context free language is closed under union operation, concatenation operation and star operation.

Comment

**Step 2** of 3

The definition of the regular expressions it can be inferred and known that regular expression consist of series of operations such as union, concatenation and star.

**Statement:** Every regular language is context free language

**Proof by using induction:**

Assume that *E* is a regular expression and there exists a CFG *G* such that $L(E)=L(G)$. The proof will be carried out by different number of operators in an expression *E*.

**Step-1:**

If $Operation(E)=0$ then $E$ is either $\phi, \in or\, a$ in $\sum$.

**Step-2:**

• If $E=\phi$: $G=(\{S\},\{\ \},P,S)$. Here $P=\{\ \}$.

• If $E=\in$: $G=(\{S\},\{\ \},P,S)$. Here $P=\{S\to\in\}$.

• If $E=a$: $G=(\{S\},\{a\},P,S)$. Here $P=\{S\to a\}$.

For every regular expression consisting of $\phi, \in or\, a$ the CFG can be written. So, every regular expression is context free.

**Thus, every regular language is context free language.**

Comment

**Step 3** of 3

The context free languages are superset of regular languages. The rules for converting the regular languages consisting of union, concatenation and Kleene closure are as follows:

• If the regular expression $E$ consists of $E_1$ and $E_2$ such that the concatenation $E=E_1E_2$, then the production can be easily expressed as $S\to E_1E_2$.

• If the regular expression $E$ consists of $E_1$ and $E_2$ such that the union $E=E_1\mid E_2$, then the production can be easily expressed as $S\to E_1\mid E_2$.

• If the regular expression $E$ consists of $E_1$ such that the Kleene closure $E=E_1^*$, then the production can be easily expressed as $E_1^*\to EE_1^*\mid\in$.

Comment

## Problem

**a.** Let C be a context-free language and R be a regular language. Prove that the language C ∩ R is context free.

**b.** Let A = {wlw ? {a, b, c}* and w contains equal numbers of a's, b's, and c's}. Use part (a) to show that A is not a CFL.

## Step-by-step solution

**Step 1** of 2

2287-2-18P AID: 824

RID: 944

a) Let $C$ be a context free language and $R$ be a regular language.

Now we have to prove that the language $C \cap R$ is a context free.

In order to prove, let us consider

$P$ be the PDA (Push Down Automata) recognizes $C$, and

$D$ be the DFA (Deterministic Finite Automata) recognizes $R$.

Now we construct a PDA that recognizes $C \cap R$ with the set of states $Q \times Q'$

where

$Q$ be the set of states of $P$ and

$Q'$ is the set of states of $D$

Here $P'$ will do what $P$ does and keep track of the states of D.

The PDA that recognizes $C \cap R$ accepts the string $w$ if and only if it stops a state $q \in F_P \times F_D$

where

$F_P$ is the states of accepts of $P$ and

$F_D$ is the states of accepts of $D$.

So $C \cap R$ is recognized by $P'$.

**Therefore, $C \cap R$ is context free.**

Comment

**Step 2** of 2

b) Given the language is

$$A = \left\{ w \mid w \in \{a,b,c\}^* \text{ and contains equal number of } a\text{'s, } b\text{'s and } c\text{'s} \right\}$$

Now we have to prove $A$ is not a CFL (Context Free Language) .

Let $R$ be the regular language $a^* b^* c^*$.

If $A$ were a CFL (Context Free Language) then $A \cap R$ would be a CFL (using the result proved above in part (a) of this problem).

Hence, inorder to prove that $A$ is not a CFL it is enough to prove that $A \cap R$ is not a CFL.

We have $A \cap R = \left\{ a^n b^n c^n \mid n \geq 0 \right\}$.

We will prove $A \cap R$ is not a CFL by taking a contradiction.

Assume that $A \cap R$ is a CFL.

Using the pumping lemma, which states thet every context-free language has a special calue called *pumping length* such that all longer strings in the language can be "pumped", let $p$ be the pumping length for $A \cap R$.

Consider a string $s = a^p b^p c^p$.

Clearly $s$ is a member of $A \cap R$ and of length at least $p$.

Now we prove that one condition of pumping lemma violated by proving $s$ cannot be pumped.

If we divide $s$ into $wxyz$, condition 2 stipulates that either $v$ or $y$ is non-empty.

Now consider one of the two cases, depending on wheather substring $v$ and $y$ contains more than one type of alphabet symbol.

1. If both $v$ and $y$ contain only one type of symbol, $v$ doesn't contain both $a$'s and $b$'s or both $b$'s and $c$'s, and the same holds for $y$. Here the string $uv^2xy^2z$ cannot contain equal number of $a$'s, $b$'s and $c$'s. Therefore it cannot be a member of $A \cap R$ which violates the first condition of the pumping lemma and thus is a contradiction to our hypothsis.

2. If either $v$ or $y$ contain more than one type of symbol $uv^2xy^2z$ may contain equal number of the three alphabet symbols but not in the correct order. Hence it cannot be a member of $A \cap R$ and thus is a contradiction to our hypothsis.

One of the above two case must occur. However, both the cases raised contradiction. This is because of our assumption $A \cap R$ is a CFL.

Hence our assumption is false and $A \cap R$ is not a CFL

**Therefore, $A$ is not a CFL.**

Comments (3)

## Problem

Let CFG G be the following grammar.

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \mathtt{b}Y \mid Y\mathtt{a}$$
$$Y \rightarrow \mathtt{b}Y \mid \mathtt{a}Y \mid \varepsilon$$

Give a simple description of L(G) in English. Use that description to give a CFG for $\overline{L(G)}$, the complement of L(G).

## Step-by-step solution

**Step 1** of 2

**Consider the context free grammar (CFG) $G$ is as follows:**

$S \rightarrow aSb \mid bY \mid Ya$
$Y \rightarrow bY \mid aY \mid \varepsilon$

**Language *L* ( *G* ) for the G is as follows:**

Consider the productions in the grammar

$S \rightarrow aSb$
$S \rightarrow bY$
$S \rightarrow Ya$
$Y \rightarrow bY$
$Y \rightarrow aY$
$Y \rightarrow \varepsilon$

**Case 1:**

Consider production $S \rightarrow Ya$ to derive the language.

Substitute Y with production $Y \rightarrow \varepsilon$ then

$S \rightarrow \in a$

$S \rightarrow a$

**Case 2:**

Consider production $S \rightarrow bY$ to derive the language.

Substitute Y with production $Y \rightarrow \varepsilon$ then

$S \rightarrow \in b$

$S \rightarrow b$

**Case 3:**

Consider production $S \rightarrow aSb$ to derive the language

Substitute S with production $S \rightarrow bY$ then

$S \rightarrow abYb$

Substitute Y with production $Y \rightarrow bY$ then

$S \rightarrow abbYb$

Substitute Y with production $Y \rightarrow \varepsilon$ then

$S \rightarrow abb \in b$

$S \rightarrow abbb$

**Case 4:**

Consider production $S \rightarrow bY$ to derive the language.

Substitute Y with production $Y \rightarrow bY$ then

$S \rightarrow bbY$

Substitute Y with production $Y \rightarrow \varepsilon$ then

$S \rightarrow bb \in$

$S \rightarrow bb$

Therefore form the Case 1, Case 2, Case 3 and Case 4 the language obtained is as follows:

*L*(*G*) = {*a, b, abbb, bb…*}

Using the grammar *G*, many more strings can be generated.

Comment

**Step 2** of 2

**Description of the *L* ( *G* ) is as follows:**

The grammar G generates a language *L*(*G*) consists of the strings which are described as follows:

• Strings with consecutive number of ***a***'s with a length ranging from 1 to infinity.

• Strings with consecutive number of ***b*'s** with a length ranging from 1 to infinity.

• String with start symbol ***a*** followed by number of ***b*'s**.

• Strings with start symbol ***b*** followed by number of ***a*'s.**

• Strings with ***a*** as start symbol and ***b*** as end symbol.

• Strings with ***b*** as start symbol and ***a*** as end symbol.

• Strings that contains the same start and end symbols. For example, $aba,\ bab$ etc.

From the above description as *L*(*G*) is generating all the possible combination of *a*'s and *b*'s except $a^i b^i \ where\ i \geq 0$. The *L*(*G*) does not produce strings like $\in, ab,\ aabb,\ aaabbb\ \ldots$

The complements of $L(G)$ i.e. $\overline{L(G)}$ = $\{\in, ab,\ aabb,\ aaabbb\ \ldots\}$

The grammar for $\overline{L(G)}$ is $a^i b^i \ where\ i \geq 0$.

Therefore, the CFG $G'$ for $\overline{L(G)}$ is as follows:

$S \rightarrow aSb \mid \in$

Comment

## Problem

Let A/B = {wl wx ? A for some x ? B}. Show that if A is context free and B is regular, then A/B is context free.

## Step-by-step solution

**Step 1** of 1

From the problem statement assume a language A is context free and a language B is regular.

From the theorem 2.20: A language is a context free language if and only if some push down automaton recognizes it.

From the corollary 1.40: A language is regular if and only if some nondeterministic finite automaton (NFA) recognizes it.

To prove that A/B is context free then it must be recognized by some push down automaton.

The proof is as follows:

Firstly construct a push down automata $P_A$ for the context free language $A$

Consider the following pushdown automata $P_A$:

$P_A=\left(Q_A,\Sigma,\Gamma,\delta_A,q_A,F_A\right)$

and a NFA M for the regular language $B$.

Consider the following NFA *M*:

$M=\left(Q_B,\Sigma,\delta_B,q_B,F_B\right)$

Assume $P_{A/B}$ is the push down automaton that recognizes the Context free language *A/B.*

$P_{A/B}=\left(Q^{A/B},\Sigma,\Gamma^{A/B},\delta_{A/B},q_{A/B},F_{A/B}\right)$

• Now $P_{A/B}$ will read the prefix **w** of the input string.

• $P_{A/B}$ will guess that it has reached to the end of input string **w** at a non-deterministically chosen point;

• $P_{A/B}$ will behave like $P_A$ and *M* running concurrently, except that it will guess the input string **x,** rather than actually reading it as input.

• If it is feasible in this way to concurrently to reach an accepting state of both $P_A$ and *M* then $P_{A/B}$ accepts.

• Note that there is no reason why the stack would have to be empty at the point where $P_{A/B}$ begins the guessing phase.

• So it is essential for $P_{A/B}$ to carry on *modeling* $P_A$ in order to properly account for the stack contents.

$P_{A/B}$ is defined as follows:

• $Q^{A/B}=Q_A\ (Q_A \times Q_B)$

• $\Gamma^{A/B} = \Gamma$

• $q_{A/B}=q_0 P_A$ where $q_0=q_A=q_B$

• $F_{A/B}=F_A\times F_B$

• $\delta_{A/B}$ is defined as follows: For $Q_A \in Q_A$ (*i.e. if* $P_{A/B}$ is the initial phase):

$$\delta_{A/B}(q_A,a,u)=\begin{cases}\delta_A(q_A,a,u), & \text{if a}\in\Sigma,\\ \delta_A(q_A,\in,u)\cup\{(q_A,q_{B,0}),\in\} & \text{if a}=\in.\end{cases}$$

For $(q_A,q_B)\in Q_A \times Q_B)$ (is the guessing phase):

$$\delta_{A/B}((q_A,q_B),a,u)=\begin{cases}\phi, & \text{if a}\in\Sigma,\\ \cup_{b\in\Sigma_\in}\{((r_A,r_B),v):(r_A,v)\in\delta_A(q_A,b,u)\text{ and r}_A\in\ \delta_B(q_B,b)\}, & \text{if a}=\in.\end{cases}$$

• Therefore it can be claimed that $P_{A/B}$ accepts **w** if and only if there occurs a string **x** such that $P_A$ accepts **wx** and *M* accepts **x**.

• For instance an acceptance calculation of $P_{A/B}$ on input **w**, all of **w** must be read during the 1st stage.

• The input symbols b that are predicted through the 2nd stage determine a string **x** that is recognized M and is such that **wx** is recognized by $P_A$.

• Contrariwise, if w is a string with the property that **wx** belongs to A for some x belong to B, then there is an acceptance calculation of $P_{A/B}$ in which w is read through the 1st stage, and the input x is predicted in the 2nd stage.

• In this instance the $P_A$ components of the states determine an acceptance calculation on $P_A$ on input **wx** and the *M*- components of the states determine an acceptance calculation of B on input **x**.

Hence, it is proved A/B is context free language by using $P_{A/B}$ from the above discussion.

## Problem

Let ∑ = {a,b}. Give a CFG generating the language of strings with twice as many a's as b's. Prove that your grammar is correct.

## Step-by-step solution

**Step 1** of 4

Consider the language *L* that generates strings with twice as many $a\text{'s as } b\text{'s}$ over the input alphabet $\Sigma = \{a,b\}$. The language does not care about the order in which the symbols $a\text{'s and } b\text{'s}$ occur.

Comment

**Step 2** of 4

The CFG for the language *L* is as follows:

$$T \to Saab \mid aSab \mid aaSb \mid aabS \mid Saba \mid aSba \mid abSa \mid abaS \mid Sbaa \mid bSaa \mid baSa \mid baaS$$
$$S \to T \mid \varepsilon$$

Comments (5)

**Step 3** of 4

Now, prove the grammar is correct using the induction.

The smallest possible strings that are generated by the grammar are $\{aab, aba, baa\}$. Let $w$ be the string from the set of smallest possible strings, such that $f_a(w) = 2f_b(w)$, where $f_a(w)$ is the number of $a\text{'s}$ in the string $w$ and $f_b(w)$ is the number of $b\text{'s}$ in the string $w$. Hence, all the smallest possible strings have twice as many $a\text{'s as } b\text{'s}$.

$$f_a(w_n) = 2f_b(w_n) \qquad .....(1)$$

(where $w_n$ represents the string of length $n$)

Comment

**Step 4** of 4

Now, show that $f_a(w_{n+1}) = 2f_b(w_{n+1})$ holds.

Obtain the string $w_{n+1}$ by inserting any of the strings $\{\varepsilon, aab, aba, baa\}$ in to $w_n$. The insertions may result in addition of 0 *a*'s and 0 *b*'s or 2 *a*'s and 1 *b*.

Case 1:

When $\varepsilon$ is inserted, (inserting $0\ a\text{'s and } 0\ b\text{'s}$)

$$f_a(w_{n+1}) = f_a(w_n) + f_a(\varepsilon) = f_a(w_n) + 0 = f_a(w_n) \qquad .....(2)$$
$$f_b(w_{n+1}) = f_b(w_n) + f_b(\varepsilon) = f_b(w_n) + 0 = f_b(w_n) \qquad .....(3)$$

Now, substitute (2) and (3) in (1).

$$f_a(w_n) = 2f_b(w_n)$$
$$f_a(w_{n+1}) = 2f_b(w_{n+1})$$

Case 2:

When $aab \text{ or } aba \text{ or } baa$ is inserted, (inserting $2\ a\text{'s and } 1\ b$)

$$f_a(w_{n+1}) = f_a(w_n) + f_a(w) = f_a(w_n) + 2 \qquad .....(4)$$
$$f_b(w_{n+1}) = f_b(w_n) + f_b(w) = f_b(w_n) + 1 \qquad .....(5)$$
$$(\text{where } w \text{ is } aab \text{ or } aba \text{ or } baa)$$

Using (4),

$$\begin{aligned} f_a(w_{n+1}) &= f_a(w_n)+2 && (\text{from } (4)) \\ &= 2f_b(w_n)+2 && (\text{from } (1)) \\ &= 2(f_b(w_n)+1) \\ f_a(w_{n+1}) &= 2f_b(w_{n+1}) && (\text{from } (5)) \end{aligned}$$

From both the cases, it is proved that $f_a(w_{n+1}) = 2f_b(w_{n+1})$.

Hence, from the principle of mathematical induction the grammar is correct.

**Therefore, the CFG generates the language of strings with twice as many *a* 's as *b* 's.**

Comments (1)

## Problem

Let
$C = \{x\#y \mid x, y \in \{0,1\}^* \text{ and } x \neq y\}$. Show that $C$ is a context-free language.

## Step-by-step solution

**Step 1** of 2

The grammar *C* is defined as follows:

$C = \{x\#y \mid x, y \in \{0, 1\}^* \text{ and } x \neq y\}$.

Comment

**Step 2** of 2

• Given that a string $x\#y$ is in language *C* if and only if $x \neq y$ or strings *x* and *y* vary at some specific position; Such as for *i*-index value of *x* is different from the character value of *y*.

• It is very easy to form a Context free grammar which produce all the strings of the form $x\#y$ with $x \neq y$.

The CFG grammar is as follows:

$S \to A\#B \mid B\#A$
$A \to TAT \mid 0$
$B \to TBT \mid 1$
$T \to 0 \mid 1$

As the grammar for *C* is defined in terms of CFG. The language produces a string that contains $x\#y$, and *x* and *y* are different character for same index position.

**Hence, it is proved that *C* is Context Free Language.**

Comments (4)

## Problem

Let $D = \{xy | x, y \in \{0,1\}^* \text{ and } |x| = |y| \text{ but } x \neq y\}.$ Show that D is a context-free language.

## Step-by-step solution

**Step 1** of 2

By the definition of **Context free language,** for showing that the language $D$ is a CFL i.e. context free language, generate a context free grammar CFG $G$.

Consider the following grammar $G$:

$S \to AB | BA$

$A \to 0 | 0A0 | 0A1 | 1A0 | 1A1$

$B \to 1 | 0B0 | 0B1 | 1B0 | 1B1$

The given grammar L (G) generates the language in the form $w_1 x w_2 v_1 y v_2$, where $|w_1| = |w_2| = k, |v_1| = |v_2| = l, x \neq y$ for $\Sigma = (0,1)^*$.

Comments (2)

**Step 2** of 2

• By the definition, any language which is generated by a context-free grammar is termed as a context-free language.

• The grammar generated above is a Context Free Grammar. The language D can be generated using the above context free grammar G as follows:

• A string is in D iff it can be written as $xy$ with $|x| = |y|$ s.t. for some $i$, the $ith$ character of $x$ and $y$ are different from one another. The above grammar can be used to obtain the required string by generating the $ith$ characters and filling up with the remaining characters.

• The generated language $w_1 x w_2 v_1 y v_2$ can be subjected to nested induction over $k$ and $l$ with case distinction over pairs $(x, y)$.

• Now, w2 and v1 can exchange symbols because both carry symbols that are independent of the rest of the string.

• Therefore, x and y in their respective half can have the same position, which implies L (G) = L because G doesn't impose any restrictions on its language.

Hence, the given language **D** is **context free language**.

Comment

## Problem

Let E = {$a^i b^j$ | i ≠ j and $2^i \neq {}^j$}. Show that E is a context-free language.

## Step-by-step solution

**Step 1** of 4

Given language E is defined as follows:

$E = \{a^i b^j \mid \mathrm{i} \neq \mathrm{j} \text{ and } 2\mathrm{i} \neq j\}$

In order to show that the language $E$ as a context free language.

Consider the language $E$ as the language of the following three languages:

$E_1 = \{a^i b^j \mid j < i\}$

$E_2 = \{a^i b^j \mid i < j < 2i\}$

$E_3 = \{a^i b^j \mid j > 2i\}$

Since context free languages are closed under the union operation. So for proving that the language $E$ is context free user has to show that the all three languages which are written above are closed under union operation.

Comment

**Step 2** of 4

For the language $E_1$ build the grammar as follows:

$S \to aAB$

$A \to aA \mid \in$

$B \to aBb \mid \in$

The non-terminal symbol $B$ generates $a^i b^j$ for $j \geq 0$ and the non-terminal symbol $A$ generates $a^i$ for $i \geq 0$. The starting non terminal symbol $S$ always includes an ***a*** so that user can conclude that any string which is generated by using above grammar has more $a's$ than $b's$.

Conversely, if the string $w$ belongs to language $E_1$, then $w$ can be written as $w = aa^{i-j-1}a^j b^j$ and assume that A generates $a^{i-j-1}$ and $B$ generates $a^j b^j$.

Comment

**Step 3** of 4

For the language $E_3$ build the grammar as follows:

$S \to ABb$

$A \to aAbb \mid \in$

$B \to Bb \mid \in$

Now the non-terminal symbol $A$ generates $a^i b^{2i}$ for $i \geq 0$ and the symbol $B$ generates $b^j$ for $j \geq 0$. Assume that this grammar derives the string $w$. Now suppose $s$ is used for storing the total number of time user replaces $A$ with $aAbb$ and $t$ is used for storing the total number of times user replaces $B$ with $Bb$. Then $w = a^s b^{2s+t+1}$ as $s, t \geq 0,\ 2s + t + 1 > 2s$ (using *j>2i* assume *i=s* and *2s+t+1=j*) and $w$ belongs to $E_3$.

Conversely, if the string $w$ belongs to language $E_3$, then $w$ can be written as $w = a^i b^{2i} b^{j-2i-1} b$ and assume that $A$ derives $a^i b^{2i}$ and the non-terminal symbol $B$ generate $b^{j-2i-1}$.

Comment

**Step 4** of 4

For the language $E_2$ build the grammar as follows:

$S \to aAb$

$A \to aAb | aAbb | abb$

Assume this grammar generates the string $w$ and assume that s is used for storing the total number of times the rule $A \to aAb$ is used and is used for storing the total number of times the rule $A \to aAbb$. But $S \to aAb$ and $A \to abb$ are used exactly once in the derivation of the string *w.* Then $w = a^{s+t+2}b^{s+2t+3}$ by assuming $i = s+t+2, j = s+2t+3$, here $s,t \geq 0$ and user has $i = s+t+2 < s+2t+3 = j$ and $2i = 2s+2t+4 > s+2t+3 = j$. Hence the string $w$ belongs to $E_2$.

Conversely, if the string $w$ belongs to language $E_2$ then $w$ can be written as $w = a^i b^j$. Assume that $s = j-i-1$ and $t = 2i-j-1$. As $i < j < 2i$ and $s,t \geq 0$ this grammar generates the string $w$ by using the rule $S \to aAb$ *s* times and $A \to aAbb$ $t$ times.

Since $E_1$, $E_2$ and $E_3$ languages are closed under union operation.

**Therefore, the language $E$ is context free language.**

Comment

## Problem

For any language A, let *SUFFIX*(A) = {vl uv ∑ A for some string u}. Show that the class of context-free languages is closed under the *SUFFIX* operation.

## Step-by-step solution

**Step 1** of 2

For any language *A*, it's suffix is defined as, $SUFFIX(A) = \{v \mid uv \in A \text{ for some string u}\}$. In order to prove that the CFLs are closed under *SUFFIX* operation, the push down automata (PDA) can be constructed or context free grammar (CFG) can be written for *SUFFIX* operation.

Comment

**Step 2** of 2

To prove the context free languages closed under context free languages, take a context free language *A*. There exists a PDA and CFG for the language *A* since it is context free. Construct the PDA for *SUFFIX* operation of *A*. Let the PDA for the language *A* be *P*. The PDA for *SUFFIX*(*A*) be *M*. Following is the procedure to construct a PDA *M*.

• Create a copy of the PDA *P* and name it as *Q*. The PDA *Q* has the same transitions as *P* as it is a replica of *P*. The PDAs *P* and *Q* combined to form the PDA *M*.

• Modify the input part of transition in *Q* to $\varepsilon$ without changing the stack symbol. If the input transition has $0,1 \to \varepsilon$, modify it to $\varepsilon,1 \to \varepsilon$. The input in the transition $0,1 \to \varepsilon$ is 0 and it is changed $\varepsilon,1 \to \varepsilon$ where the stack symbol $\varepsilon$ is unchanged. In this step, just change the input part of each transition irrespective of the stack symbol.

• For each state in PDA *Q*, add a new transition $\varepsilon,\varepsilon \to \varepsilon$ to the corresponding state in PDA *P*. This means, for the input $\varepsilon$ and stack symbol $\varepsilon$, the top of the stack will be $\varepsilon$. This step simply connects two PDAs.

• The start state of PDA *Q* should be the start state of the whole PDA *M*. Thus, the PDA *M* is the combination of two PDAs *Q* and *P*.

The PDA *M* simply ignores the alphabet of *u* and starts functioning when it identifies the first alphabet of *v* from which the second part of the PDA *M* (i.e., *P*) accepts the substring *v* (i.e., suffix). Thus, all the suffixes of the string belong to language *A* will be accepted by the PDA *M*.

Therefore, the CFLs are closed under *SUFFIX* operation.

Comment

## Problem

Show that if G is a CFG in Chomsky normal form, then for any string w ∑ L(G) of length n ≥ 1, exactly 2n – 1 steps are required for any derivation of w.

## Step-by-step solution

**Step 1** of 1

Given that *G* is a CFG in Chomsky Normal Form (CNF).

The length of the string $w \in L(G)$ is n ≥ 1 for the string *w.*

It is required to show that exactly 2*n*-1 steps are required for the derivation of string $w$.

It can be proved applying the **induction method** by on the string w of length *n.*

**For $n = 1$ :** Consider a string "**a**" of length 1 in Chomsky normal form, so the valid derivation for this will be $S \to a$, where $a \in \sum$ and $S$ is starting symbol.

The number of steps can be obtained as follows:

$$\begin{aligned} 2n-1 &= 2(1)-1 \\ &= 2-1 \\ &= 1 \end{aligned}$$

Hence it is true that 2n-1 (for n=1) steps are required to derive a string **a**.

**For $n = k$ :** Take a string of length $k \geq n$ in Chomsky normal form, so valid derivation for this will take $2k-1$ steps.

The number of steps can be obtained as follows:

$$\begin{aligned} 2n-1 &= 2(k)-1 \\ &= 2k-1 \end{aligned}$$

Assume a string of length at most k≥1 terminal symbols and it has a string of length

n=$k+1$ is in Chomsky Normal Form

Since n>1, Consider a language as follows in CNF where derivation starts with start symbols S:

$$\begin{aligned} &S \to BC \\ &B \to {}^*x \\ &C \to {}^*y \end{aligned}$$

So length of the string starting with start symbol S is l*w*l = xy where lxl > 0 and l y l > 0.

Using the inductive hypothesis, for the above language in CNF the length of any derivation of string *w* must be

$$1+(2|x|-1)+(2|y|-1) = 2|x|+2|y|+1-1-1 = 2(|x|+|y|)-1$$

Here n=$|x|+|y|$.

Since $B \to {}^*x$ has a length of *lxl* and $C \to {}^*y$ has a length of *lyl.*

Hence, it is proved that it requires **2n-1** steps required for the derivation of string

$w \in L(G)$ in Chomsky Normal Form(CNF).

Comment

Let $G = (V, \Sigma, R, \langle \text{STMT} \rangle)$ be the following grammar.

$$\begin{aligned}
\langle \text{STMT} \rangle &\to \langle \text{ASSIGN} \rangle \mid \langle \text{IF-THEN} \rangle \mid \langle \text{IF-THEN-ELSE} \rangle \\
\langle \text{IF-THEN} \rangle &\to \texttt{if condition then } \langle \text{STMT} \rangle \\
\langle \text{IF-THEN-ELSE} \rangle &\to \texttt{if condition then } \langle \text{STMT} \rangle \texttt{ else } \langle \text{STMT} \rangle \\
\langle \text{ASSIGN} \rangle &\to \texttt{a:=1}
\end{aligned}$$

$$\Sigma = \{\texttt{if}, \texttt{condition}, \texttt{then}, \texttt{else}, \texttt{a:=1}\}$$

$$V = \{\langle \text{STMT} \rangle, \langle \text{IF-THEN} \rangle, \langle \text{IF-THEN-ELSE} \rangle, \langle \text{ASSIGN} \rangle\}$$

G is a natural-looking grammar for a fragment of a programming language, but G is ambiguous.

**a.** Show that G is ambiguous.

**b.** Give a new unambiguous grammar for the same language.

**Step-by-step solution**

**Step 1** of 3

**Ambiguous and unambiguous grammar**

a.

Consider $G = (V, \Sigma, R, \langle \text{STMT} \rangle)$

To show that G is ambiguous

It is required to find out two leftmost derivations:

$$\begin{aligned}
&\langle \text{Stmt} \rangle \\
&\Rightarrow \langle \text{if-then} \rangle \\
&\Rightarrow \text{if condition then } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then } \langle \text{if-then-else} \rangle \\
&\Rightarrow \text{if condition then if condition then } \langle \text{Stmt} \rangle \text{ else } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then if condition then } \langle \text{Assign} \rangle \text{ else } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then if condition then a:=1 else } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then if condition then a:=1 else } \langle \text{Assign} \rangle \\
&\Rightarrow \text{if condition then if condition then a:=1 else a:=1}
\end{aligned}$$

Now the second derivation will be:

$$\begin{aligned}
&\langle \text{Stmt} \rangle \\
&\Rightarrow \langle \text{if-then-else} \rangle \\
&\Rightarrow \text{if condition then } \langle \text{Stmt} \rangle \text{ else } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then } \langle \text{if-then} \rangle \text{ else } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then if condition then } \langle \text{Stmt} \rangle \text{ else } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then if condition then } \langle \text{Assign} \rangle \text{ else } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then if condition then a:=1 else } \langle \text{Stmt} \rangle \\
&\Rightarrow \text{if condition then if condition then a:=1 else } \langle \text{Assign} \rangle \\
&\Rightarrow \text{if condition then if condition then a:=1 else a:=1}
\end{aligned}$$

Comment

**Step 2** of 3

In both cases when it takes “if-then” or “if-then else” result is same. Hence both have same left derivation and it is ambiguous.

Comment

**Step 3** of 3

For making unambiguous grammar, it is required to make correct interpretation of the above two. For that when "if-then-else" is derived, it should not allow then part for deriving "if-then”. So introduce a new variable $\langle \text{Stmt} \rangle$ the new grammar is:

$$\langle \text{Stmt} \rangle \Rightarrow \langle \text{if-then} \rangle \mid \langle \text{Assign} \rangle \mid \langle \text{if-then-else} \rangle$$
$$\langle \text{if-then-else} \rangle \Rightarrow \text{ if condition then } \langle \text{Stmt1} \rangle \text{ else } \langle \text{Stmt} \rangle$$
$$\langle \text{Stmt1} \rangle \Rightarrow \langle \text{if-then-else} \rangle \mid \langle \text{Assign} \rangle$$
$$\langle \text{if-then} \rangle \Rightarrow \text{if condition then } \langle \text{Stmt} \rangle$$
$$\langle \text{Assign} \rangle \Rightarrow \text{a:=1}$$

Comments (2)

## Problem

Give unambiguous CFGs for the following languages.

**a.** {wl in every prefix of w the number of a's is at least the number of b's}

**b.** {wl the number of a's and the number of b's in w are equal}

**c.** {wl the number of a's is at least the number of b's in w}

## Step-by-step solution

**Step 1** of 4

First step is finding a grammar for finding an unambiguous CFG whether it is ambiguous or unambiguous.

• Grammar for the given problem is

$$S \to SS \mid (S) \mid \varepsilon$$
$$S \to a \mid b$$

• But the given grammar is ambiguous as there is more than one parse tree is possible for the above grammar.

• After finding the grammar finds that whether it is ambiguous or not.

• If the grammar is ambiguous then set some priority rules for selecting derivation tree for the grammar and after that set production rules for the grammar.

• Now after setting some priority rules the grammar can be:

$$S \to (S)S \mid \varepsilon$$
$$S \to a \mid b$$

• But this is also wrong grammar as in this grammar for every prefix of any input string w the number of a's does not at least the number of b's.

• Final grammar in which for every prefix of any input string *w* the number of a's is at least the number of b's is as shown:

$$S \to (S)S \mid (S \mid \varepsilon$$
$$S \to a \mid b$$

Comment

**Step 2** of 4

a)

The input alphabet *a* is prefixes at least the number of b's that means if the length of the input string is one then that must be a.

For the two or more inputs the possible strings are *aa*, *ab*, *aaa*, *aab* and so on.

So, the unambiguous grammar (CFG) is given below:

$$S \to aS \mid a \mid b \mid \varepsilon$$

The production for a regular expression *aaab*.

$$\begin{aligned} S &\to aS \\ &\to aaS \\ &\to aaaS \\ &\to aaab \end{aligned}$$

This grammar is an unambiguous grammar because there is no possible way to get same regular expression.

Comments (1)

**Step 3** of 4

b)

Consider the following unambiguous CFG:

$S \to aSb | bSa | \varepsilon$

In the above grammar, produ ion rules $S \to aSb$ and $S \to bSa$ generates the equal number of terminals a's and b's.

$$\begin{aligned} S &\to aSb \\ &\to a(aSb)b \\ &\to aa(aSb)bb \\ &\to aaabbb \end{aligned}$$

Comments (5)

**Step 4** of 4

c)

In the given question, the number *a's* is at least the number of *b's* that means for a single character input *a* must be the input alphabet.

Consider the following unambiguous CFG:

$S \to aSb | bSa | aS | \varepsilon$

• In the above grammar, production rules $S \to aSb$ and $S \to bSa$ generates the equal number of terminals *a's* and *b's*.

• The production rule $S \to aS$ is used for generating the terminal *a's* as many as user wants.

• The production rule $S \to \varepsilon$ is used for generating the equal number of terminals *a's* and *b's*.

$$\begin{aligned} S &\to aSb | bSa | aS | \varepsilon \\ &\to a(aSb)b \\ &\to aa(bSa)bb \\ &\to aababb \end{aligned}$$

Comments (3)

## Problem

Show that the language A in Exercise 2.9 is inherently ambiguous.

## Step-by-step solution

**Step 1** of 4

If a string w has two or more different derivations, it is ambiguous. The ambiguous grammar will generate some string ambiguously.

Some context free grammars can only be generated by the ambiguous grammars. Such languages are called ***inherently ambiguous***.

Comment

**Step 2** of 4

Given context-free grammar that generates the language

$$A = \left\{ a^i b^j c^k \mid i = j \text{ or } j = k \text{ where } i, j, k \geq o \right\}$$

Let $G$ be the context-free grammar generating the language $A$ and $G$ is an ambiguous grammar. Now, show that $A$ is inherently ambiguous.

Let $p$ be the pumping length of the context-free grammar $G$ given by the pumping lemma.

Let $k$ be an integer such that $k = p! = p(p-1)(p-2)...1$.

Let $s$ be a string such that $s = a^k b^k c^k$.

To show that $A$ is inherently ambiguous, it is enough to show that $s$ has two or more different leftmost derivations. It is required to know that those parse trees are equivalent to derivations and $s$ has two parse trees.

Comment

**Step 3** of 4

Let $s_1 = a^k b^p c^p$ and $s_2 = a^p b^p c^k$ are two strings in $A$ with parse trees $\tau_1$ and $\tau_2$ respectively.

Take $\tau_1$ and remove all nodes that have only $a's$ below them. Since there are $k\ a's$, the sub tree will have only $p\ b's$ and $p\ c's$, in total having $2p$ leaves.

Because $2p > p$ it is known that the sub tree contains a path with repeated variable. Let $R$ be the repeated variable. Divide $s$ into $uvxyz$ using $R$ and $R$ can be chosen such that $|vxy| \leq p$. Each of the strings $v$ and $y$ can contain only one of the symbols, if not $uv^2xy^2z$ will not belong to $A$.

Also $y$ contain no $a's$, because $R$ was on a path to $a, b$ or $c$. Hence $v$ must be $b's$ and $y$ must be $c's$ and they must have an equal length $l$. Consider $d = k/l$ (which must be an integer since $l \leq p$ and $k = p!$) the string $s = vu^d xy^d$ has a parse tree where most $b's$ have a parent node in common with $c's$ but not with $a's$.

Comment

**Step 4** of 4

Similarly, consider $\tau_2$, the parse tree is obtained for $s$ $b's$ having a parent node in common with $a's$ but not with $c's$.

So $s$ has two distinct parse trees. Hence $s$ has two or more different leftmost derivations and $G$ can be generated by the ambiguous grammars only.

**Therefore, the language $A$ is inherently ambiguous.**

Comment

## Problem

Use the pumping lemma to show that the following languages are not context free.

**a.** $\{0^n 1^n 0^n 1^n \mid n \geq 0\}$

A **b.** $\{0^n \# 0^{2n} \# 0^{3n} \mid n \geq 0\}$

A **c.** $\{w\#t \mid w \text{ is a substring of } t, \text{ where } w, t \in \{\mathtt{a}, \mathtt{b}\}^*\}$

**d.** $\{t_1 \# t_2 \# \cdots \# t_k \mid k \geq 2, \text{ each } t_i \in \{\mathtt{a}, \mathtt{b}\}^*, \text{ and } t_i = t_j \text{ for some } i \neq j\}$

## Step-by-step solution

**Step 1** of 5

a) Consider the language $B = \{0^n 1^n 0^n 1^n \mid n \geq 0\}$.

Let $p$ be the pumping length of $B$ given by the pumping lemma.

To show that $B$ is not a CFL, it is enough to show that a string $s = 0^p 1^p 0^p 1^p$ cannot be pumped.

Consider $s$ is of the form $uvxyz$.

• If both $v$ and $y$ contain at most one type of alphabet symbol, the string will be of the form $uv^2xy^2z$ runs of $0\text{'s and } 1\text{'s}$ of unequal length. Hence the string $s$ cannot be a member of $B$.

• If either $v$ or $y$ contains more than one type of alphabet symbol, the string will be of the form $uv^2xy^2z$ which does not contain the symbols in correct order. Hence the string $s$ cannot be a member of $B$.

Since the string $s$ cannot be pumped without violating the pumping lemma condition, $B$ is not a CFL (context-free language).

Comment

**Step 2** of 5

b) Consider the language $B = \{0^n \# 0^{2n} \# 0^{3n} \mid n \geq 0\}$.

Let $p$ be the pumping length of $B$ given by the pumping lemma.

To show that $B$ is not a CFL, it enough to show that a string $s = 0^p \# 0^{2p} \# 0^{3p}$ cannot be pumped.

Consider $s$ is of the form $uvxyz$.

Neither $v$ nor $y$ can contains $\#$, otherwise $uv^2xy^2z$ contains more than two $\#\text{s}$. If the string $s$ is divided into three segments by $\#\text{'s}$ at least one of the segments $0^p, 0^{2p}$ and $0^{3p}$ is not contained within either $v$ or $y$.

Because the length ratio of the segments is not maintained as 1:2:3, $xv^2wy^2z$ is not in $B$.

Hence the string $s$ cannot be a member of $B$.

Since the string $s$ cannot be pumped without violating the pumping lemma condition, $B$ is not a CFL (context-free language).

Comment

**Step 3** of 5

c) Consider the language $B = \{w\#t \mid w \text{ is a substring of } t, \text{ where } w, t \in \{a, b\}^*\}$.

Let $p$ be the pumping length of $B$ given by the pumping lemma.

To show that $B$ is not a CFL, it enough to show that a string $s = a^p b^p \# a^p b^p$ cannot be pumped.

Consider $s$ is of the form $uvxyz$.

• Neither $v$ nor $y$ can contair $\#$, otherwise $uv^0xy^0z$ does not contain $\#$. Hence the string $s$ cannot be a member of $B$.

• If both $v$ and $y$ are nonempty and occur on the left-hand side of $\#$, the string $uv^2xy^2z$ is longer on the left-hand side of $\#$. Hence the string $s$ cannot be a member of $B$.

• Similarly, if both $v$ and $y$ are nonempty and occure on the right-hand side of $\#$, the string $uv^0xy^0z$ is longer on the rifht hand side of $\#$. Hence the string $s$ cannot be a member of $B$.

• If only one of $v$ and $y$ is nonempty we can treat them as if both occurred on the same side of $\#$. Hence the string $s$ cannot be a member of $B$.

• In the remaining case if both $v$ and $y$ are nonempty and include the $\#$, then by the third pumping lemma condition $|vxy| \le p$, we have $v$ consists of $b$'s and $y$ consists of $a$'s. Hence $uv^2xy^2z$ contains more $b$'s on the left-hand side of the $\#$. Hence the string $s$ cannot be a member of $B$.

Since the string $s$ cannot be pumped without violating the pumping lemma condition, $B$ is not a CFL (context-free language).

Comments (3)

**Step 4** of 5

d) Consider the language

$$B = \left\{ t_1 \# t_2 \# \cdots \# t_k \mid k \ge 2,\ t_i \in \{a,b\}^*, \text{ and } t_i = t_j \text{ for some } i \ne j \right\}.$$

Let $p$ be the pumping length of $B$ given by the pumping lemma.

The $t_i$ can be equal to $t_j$ for different *i* and *j* values. Hence, the same terms will be appeared in the string *s* separated by *#*.

For example, if the *k* value is 2 the string *s* can be *ab#ab.* The string *s* has the same term *ab* for different *k* values separated by *#*. The language generates the strings that contains same terms comprised of *a, b* separated by *#*. The strings that can be generated from the language *B* are *ab#ab, b#b#b, aba#aba#aba#aba, …* etc.

To show that $B$ is not a CFL, it enough to show that a string $s = a^p b^p \# a^p b^p$ cannot be pumped.

Consider $s$ is of the form $uvxyz$.

• Neither $v$ nor $y$ can contains $\#$, otherwise $uv^0xy^0z$ does not contain $\#$. Hence the string $s$ cannot be a member of $B$.

• If only one of $v$ and $y$ is nonempty we can treat them as if both occurred on the same side of $\#$. Hence the string $s$ cannot be a member of $B$.

• If both $v$ and $y$ are nonempty and occur on the left-hand side of $\#$, the string $uv^2xy^2z$ is longer on the left-hand side of $\#$. Hence the string $s$ cannot be a member of $B$.

• If both $v$ and $y$ are nonempty and occur on the right-hand side of $\#$, the string $uv^0xy^0z$ is longer on the rifht hand side of $\#$. Hence the string $s$ cannot be a member of $B$.

•

Comment

**Step 5** of 5

In the remaining case if both $v$ and $y$ are nonempty and include the $\#$, then by the third pumping lemma condition $|vxy| \le p$, we have $v$ consists of $b$'s and $y$ consists of $a$'s. Hence $uv^2xy^2z$ contains more $b$'s on the left-hand side of the $\#$. Hence the string $s$ cannot be a member of $B$.

Since the string $s$ cannot be pumped without violating the pumping lemma condition, $B$ is not a CFL (context-free language).

Comment

## Problem

Let B be the language of all palindromes over {0,1} containing equal numbers of 0s and 1s. Show that B is not context free.

## Step-by-step solution

**Step 1** of 3

Let $B$ be the language of all palindromes over $\{0, 1\}$ containing the equal numbers of $0$'s and $1$'s . To prove $B$ is not a context free language by taking a contradiction.
Assume that $B$ is a context free language.

Since $B$ is a context free language, then by pumping lemma, there is a number $p$ (the pumping length) where, if $s$ is any string in $B$ of length at least $p$ , then $s$ may be divided into five pieces $s = uvxyz$ satisfying the conditions

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

Comment

**Step 2** of 3

**Case 1:**

Now select a string $s = 0^n 1^{2n} 0^n$.

Clearly $s$ is a member of $B$ of length at least $p$.

Assume the value of n=2 for the string *s*.

$s = 0^2 1^4 0^2$

$s = 00111100$

The string s can be divided into *uvxyz* as follows:

$$\frac{00}{u}\ \frac{11}{v}\ \frac{1}{x}\ \frac{10}{y}\ \frac{0}{z}$$

Now apply the first condition of the pumping lemma.

for each $i \geq 0, uv^i xy^i z \in A$

For i = 2:

$$\frac{00}{u}\left(\frac{11}{v}\right)^2 \frac{1}{x}\left(\frac{10}{y}\right)^2 \frac{0}{z}$$

$$\frac{00}{u}\left(\frac{1111}{v}\right) \frac{1}{x}\left(\frac{1010}{y}\right) \frac{0}{z}$$

Assume the obtained string 00 1111 1 1010 0 as $s'$.

The obtained string $s'$ is not a palindrome after applying the first condition of pumping lemma and $s' \notin \mathrm{B}$. In the pumped string, the number of 0's and 1's is not equal.

So, the language B is not following the condition1 of the pumping lemma.

Comment

**Step 3** of 3

**Case 2:**

The same string is selected as $s = 0^n 1^{2n} 0^n$

Clearly $s$ is a member of $B$ of length at least $p$.

Assume the value of n=2 for the string s.

s=$0^2 1^4 0^2$

s=00111100

The string s can be divided into *uvxyz* as follows:

$$\frac{0}{u}\ \frac{01}{v}\ \frac{11}{x}\ \frac{1}{y}\ \frac{00}{z}$$

Now apply the first condition of the pumping lemma.

$$\text{for each } i \geq 0, uv^i xy^i z \in A$$

For i=2:

$$\frac{0}{u}\left(\frac{01}{v}\right)^2 \frac{11}{x}\left(\frac{1}{y}\right)^2 \frac{00}{z}$$

$$\frac{0}{u}\left(\frac{0101}{v}\right) \frac{11}{x}\left(\frac{11}{y}\right) \frac{00}{z}$$

Assume the obtained string 0 0101 11 11 00 as $s'$.

The obtained string $s'$ is not a palindrome after applying the first condition of pumping lemma and $s' \notin \mathrm{B}$. In the pumped string, the number of 0's and 1's is not equal.

Hence, the assumption $B$ is a context free language is wrong.

**Therefore, by the two cases it can be proved that $B$ is not a context free language.**

Comments (3)

## Problem

Let ∑ = {1, 2, 3, 4} and C = {w ? ∑*l in w, the number of 1s equals the number of 2s, and the number of 3s equals the number of 4s}. Show that C is not context free.

## Step-by-step solution

**Step 1** of 1

**Context Free Language**

Consider the language:

$$C = \left\{ \begin{array}{l} w \in \{0,1,2,3,4\}^* \mid \text{in w the number of 1s and the number of 2s} \\ \text{are equal,the number of 3s and the number of 4s are equal.} \end{array} \right\}$$

On the contrary consider C is context free. So, C has a pumping length p.

Take $s = 1^p 3^p 2^p 4^p \in C$ with $|s| > p$.

Therefore, there exist $uvxyz$ such that

(a) $uv^i xy^i z \in C$ for all $i \geq 0$ ……(1)

(b) $vy > 0$ ……(2)

(c) $vxy \leq p$ ……(3)

Now it has to prove all the cases by contradiction, no matter what the value of $uvxyz$.

**Case 1:** If $vxy$ contains a $1$. Then $uv^2xy^2z \notin C$, since it cannot be same number of $1$s and $2$s. Hence due to equation $(3)$, $vxy$ cannot contain any $2$s.

**Case 2:** If $vxy$ contains a $2$. Then $uv^2xy^2z \notin C$, since it cannot be same number of $1$s and $2$s. Hence due to equation $(3)$, $vxy$ cannot contain any $1$s.

**Case 3:** If $vxy$ contains a $3$. Then $uv^2xy^2z \notin C$, since it cannot be same number of $3$s and $4$s. Hence due to equation $(3)$, $vxy$ cannot contain any $4$s.

**Case 4:** If $vxy$ contains a $4$. Then $uv^2xy^2z \notin C$, since it cannot be same number of $3$s and $4$s. Hence due to equation $(3)$, $vxy$ cannot contain any $3$s.

Hence from equation $2$, it contradicts equation $1$ in all the cases which shows *C* is not context free language.

Comment

## Problem

Show that F = {$a^i b^j$ I i = $k_j$ for some positive integer k} is not context free.

## Step-by-step solution

**Step 1** of 3

**The language given in the question is as follows:**

Language $F = \{a^i b^j \mid i = kj \text{ for some positive integer } k\}$.

A context free language (CFL) is generated by a context free grammar. **In order to prove that a language is not a CFL, Pumping Lemma is used.**

Comment

**Step 2** of 3

**Proof that Language L is not a Context Free Language:**

Assume that *F* is CFL. Obtain a contradiction using pumping lemma to prove that the assumed statement is false.

• Let *p* be the pumping length for *F* that is guaranteed to exit by pumping lemma.

• Select string $s = a^p b^{2p} \in F$ where $k = 2$ and divide the string *s* into *uvxyz*.

• According to pumping lemma, *v* and *y* in string cannot be empty sets.

• Now, consider these two case, depending on whether substring *v* and *y* contain more than one type of alphabet symbol:

1. **Both *v* and *y* contain only one type of alphabet symbol:** In this case, both $v \text{ and } y$ does not contain mixed a's and b's. Thus, the string $uv^2xy^2z$ cannot contain equal number of a's and b's. Also, a pattern for a's and b's can be obtained that contains a relation between number of a's and b's. So, none of the conditions of lemma violates and thus it does not contradict.

For example:

Consider $uv^i xy^i z$ such that $v = a, y = bb$ and $u = x = z = \phi$. Thus, the strings generated will be $s = abb, aabbbb, aaabbbbbb...$

All the strings *s* are a member of *F*. Hence, no contradiction is obtained.

2. **Either of *v* or *y* contains more than one type of alphabet symbols**: In this case, both $v \text{ and } y$ contain mixed a's and b's. Thus, the string $uv^2xy^2z$ will contain strings with some order of $ab$ followed by some order of $ab$ again. Thus, it produces a wrong order of strings thereby producing a contradiction

For example:

Consider $uv^i xy^i z$ such that $v = ab \text{ and } y = b$ and $u = x = z = \phi$. Thus the strings generated will be $s = abb, ababbb...$

The string s = $ababbb$ is not member of *F*. This violates our assumption and thus, a contradiction is obtained.

Comments (1)

**Step 3** of 3

**The second case results in a contradiction. Hence, the assumption that *F* is context free language is false and therefore, F is not a context free language.**

Comments (2)

## Problem

Consider the language B = L(G), where G is the grammar given in Exercise 2.13. The pumping lemma for context free languages, Theorem 2.34, states the existence of a pumping length p for B. What is the minimum value of p that works in the pumping lemma? Justify your answer.

THEOREM **2.34**

**Pumping lemma for context-free languages** If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces s = uvxyz satisfying the conditions

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

When s is being divided into uvxyz, condition 2 says that either v or y is not the empty string. Otherwise the theorem would be trivially true. Condition 3 states that the pieces v, x, and y together have length at most p. This technical condition sometimes is useful in proving that certain languages are not context free.

## Step-by-step solution

**Step 1** of 3

**Given:**

The minimum value of the pumping length $p$ for the language $B = L(G)$

Comment

**Step 2** of 3

**Finding minimum length of $\mathbf{p}$ :**

The language $B = L(G)$ is defined by the

Grammar $G = (V, \Sigma, R, S)$ where $V = \{S, T, U\}$; $\Sigma = \{0, \#\}$ and the given set of rules $R$ :

$$S \to TT \mid U$$
$$T \to 0T \mid T0 \mid \#$$
$$U \to 0U00 \mid \#$$

Theorem 2.34 states that the pumping lemma $p$, for a context-free language $A$, is the minimum length of any string $s$ in $A$ such that it may be split into five pieces $s = uvxyz$. The string $s$ will also fulfill the following three conditions:

1. For all $i \geq 0$, the string $uv^i xy^i z$ is a part of the context-free language $A$.

2. The strings which are pumped, which are $v$ and $y$, cannot both be the empty string $\varepsilon$, that is $|vy| > 0$.

3. The combined length of the strings lying inside $u$ and $z$ must not be greater that the pumping length $p$. In other words $|vxy| \leq p$.

The string $s = uvxyz$ can be taken as $\#\#0$, with the substrings being as follows:

$$u = v = z = \varepsilon$$
$$x = \#\#$$
$$y = 0$$

Thus the pumping length $p$ is $|vxy| = |\#\#0| = 3$

Since, $|vy| = |\varepsilon 0| = |0| = 1 > 0$, condition 2 of the theorem holds.

Condition 3 of Theorem 2.34 is also valid as $|vxy| = |\varepsilon \#\#0| = |\#\#0| = 3 \leq p = 3$.

To meet condition 1 it has to be proven that $uv^i xy^i z \in B$ for $i \geq 0$.

The string $uv^i xy^i z$, where $u = \varepsilon, v = \varepsilon, x = \#\#, y = 0, z = \varepsilon$, can be expressed as the regular expression $\#\#0^i$.

The derivation for the case when $i = 0$ is:

$$S \Rightarrow TT \Rightarrow \#T \Rightarrow \#\#$$

So the string $s = uv^0 xy^0 z$ lies in the language $B$.

When $i > 0$ the string $s = uv^i xy^i z$ will also lie in $B$ as the derivation will be:

$$S \Rightarrow TT \Rightarrow \#T \Rightarrow \#T0 \overset{*}{\Rightarrow} \#T0^+ \Rightarrow \#\#0^+$$

Condition 1 has been proven true as $uv^i xy^i z \in B \forall i \geq 0$.

Comment

**Step 3** of 3

**Conclusion:**

Thus the string satisfies all three conditions of Theorem 2.34 for a context-free language. Therefore, **the minimum value is 3** for the pumping lemma $p$.

Comment

## Problem

Let G be a CFG in Chomsky normal form that contains b variables. Show that if G generates some string with a derivation having at least $2^b$ steps, L(G) is infinite.

## Step-by-step solution

**Step 1** of 3

**CFG:**

• A finite set of grammar rules is known as **CFG (context free grammar)**. It consisting of that is quadruple (*N*, *T*, *P*, *S*).

• Where, set of non-terminal symbol represented by *N*.

• **T** is group of terminal $N \bigcap T = \text{NULL}$.

• **P** is group of rule, $P : N \to (N \bigcup T)^*$.

• **S** is start symbol.

Comment

**Step 2** of 3

**CNF:**

• In **CNF (Chomsky normal form)** we have a restriction on the length of RHS, which is a CFG (context free grammar) is in Chomsky normal form if the productions are in the following terms:

$U \to u$
$U \to VW$

• Where *U*, *V* and *W* are non-terminals and *u* is a terminal.

Comment

**Step 3** of 3

• At most two terminal can generate in every deviation.

• In any parse string with use of *G*.

• An internal node can have at most two children.

• Parse tree with height k has at most o $2^k - 1$ internal node.

• If several string generated by *G* with a derivation taking at least $2^b$ steps.

• At least $2^b$ inner node takes parse tree of that string.

• At least consuming $b+1$ height in Parse tree.

• It exists a path from root to leaf containing $b+1$ variable.

• In this one variable occurring twice.

**Hence, user can use the technique in proof of pumping lemma to construct infinity many string which are all in L(G).**

Comments (1)

## Problem

Give an example of a language that is not context free but that acts like a CFL in the pumping lemma. Prove that your example works. (See the analogous example for regular languages in Problem 1.54.)

Problem 1.54

Consider the language $$F = \{\mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \mid i, j, k \geq 0 \text{ and if } i = 1 \text{ then } j = k\}.$$

**a.** Show that F is not regular.

**b.** Show that F acts like a regular language in the pumping lemma. In other words, give a pumping length p and demonstrate that F satisfies the three conditions of the pumping lemma for this value of p.

**c.** Explain why parts (a) and (b) do not contradict the pumping lemma.

## Step-by-step solution

**Step 1** of 6

Consider the following details:

Let $F = \{a^i b^j c^k d^m \mid i, j, k, m \geq 0 \text{ and if } i = 1 \text{ then } j = k = m\}$.

Comment

**Step 2** of 6

**Proof that F is not a context free Language (by contradiction):**

• Suppose F is context free, then $F \cap \{ab^i c^j d^k \mid i, j, k \geq 0\} = \{ab^i c^i d^i \mid i \geq 0\} = G$ is context free since $\{ab^i c^j d^k \mid i, j, k \geq 0\}$ is the language of the regular expression $ab^* c^* d^*$ which is regular, and also the intersection of a context free language with a regular language is context free.

• By showing that $G$ cannot be context free using pumping lemma, it will contradict the fact that F is context free.

Comment

**Step 3** of 6

Suppose the pumping length of $G$ is p and take $s = ab^p c^p d^p \in G$ with $|s| > p$.

There exists *u*, *v*, *x*, *y*, *z* such that $s = uvxyz$ and,

(1) $uv^n xy^n z \in G$ for all n ≥ 0,

(2) $|vy| > 0$ and

(3) $|vxy| \leq p$.

Comment

**Step 4** of 6

For any valid choice of *uvxyz* that $uv^2xy^2z \notin G$, take $i = 0$. Then,

**Case 1:** If *v* or *y* contains *a*, then $uv^2xy^2z$ will have more than one $a$ and thus is not in *G*.

**Case 2:** If *v* and *y* do not contain $a$, then from $|vxy| \leq p$, *vxy* can have at most two other symbols from *b*, *c* or *d*.

Therefore, $uv^2xy^2z$ will not have the same number of the three symbols.

**Thus, $G$ is not a CFL and therefore $F$ is also not a CFL.**

Comment

**Step 5** of 6

**Proof that F is a context free language using the pumping lemma:**

Let $p = 2$. Now for any string $s \in F$, with $|s| \geq 2$, can be written as $uvxyz$ such that

(1) $uv^n xy^n z \in G$ for all $n \geq 0$,

(2) $|vy| > 0$

(3) $|vxy| \leq p$

Now,

**Case 1:** $s = a^i b^j c^k d^m$ with $i \neq 2$ and $i + j + k + m \geq 2$.

In this case, let $u = v = x = \epsilon$, *y* is the first symbol in *s* and *z* be the remaining symbols.

Then (2) and (3) hold, and $uv^n xy^n z$ will have either zero as (*i* = 0 or *i* = 1 and *n* = 0) or more than one a followed by a string of the form $b^j c^k d^m$, so it will remain in *F* and therefore (1) holds.

**Case 2:** s = $a^2 b^j c^k d^m \in F$ for some *j*, *k*, $m \geq 0$ (so $|s| \geq 2$). Take $u = v = x = \epsilon$, $y = a^2$ (in this case *a* would not work if any *j*, *k*, *m* > 0 because there is pumping down, but if a would be taken as the first symbol then it will work), and $z = b^j c^k d^m$. Then (2) and (3) hold and $xy^i z = a^{2+2(i-1)} b^j c^k d^m \in F$, therefore, (1) also holds.

Thus, in either cases, the conditions of the pumping lemma hold.

Comment

**Step 6** of 6

**Therefore, $F$, which is not a Context Free Language, satisfies the pumping lemma.**

Comment

## Problem

Prove the following stronger form of the pumping lemma, wherein *both* pieces v and y must be nonempty when the string s is broken up.

If A is a context-free language, then there is a number k where, if s is any string in A of length at least k, then s may be divided into five pieces, s = uvxyz, satisfying the conditions:

**a.** for each $i \geq 0$, $uv^i xy^i z \in A$,

**b.** $v \neq \varepsilon$ and $y \neq \varepsilon$, and

**c.** $|vxy| \leq k$.

## Step-by-step solution

**Step 1** of 8

It has to be proven that if $A$ is a context-free language, there is a number $k$ where is a string $s$ such that $s \in A$ and $|s| \geq k$, then the string $s$ can be split into five pieces $s = uvxyz$ such that the following conditions hold:

a. For each $i \geq 0, uv^i xy^i z \in A$,

b. $v \neq \varepsilon$ and $y \neq \varepsilon$, and

c. $|vxy| \leq k$.

Comment

**Step 2** of 8

Now, define a context-free grammar $G = (V, \Sigma, R, T)$ for the context-free language $A$ such that the right-hand side of a rule $b$ has maximum number of symbols, with $b$ being at least 3 symbols large.

• When the height of parse tree is $k$ then the length of the string $s$ generated is at least $b^k$. This is as there will be at most $b$ leaves 1 step from $S$, at most $b^2$ at a 2 steps and at-most $b^k$ leaves at a depth of $k$ steps.

• Alternatively if the length of string is $b^k$ then all smallest possible parse trees for the string must be at least $k$ deep. Thus any path in a parse tree for the string $s$ will have a path that is at least $k$ terminals long. The path will start with $k-1$ variables and terminate with a single terminal.

Comment

**Step 3** of 8

The number of variables in the grammar $G$ is $|V|$. A string $s$ is taken whose length is at least $|V|+1$. Hence any of the smallest possible parse trees will have a depth that is at least $b^{|V|+1}$ steps. Or in other words all paths will be at least $|V|+1$ long, which will start with $|V|+1$ variables and end with a terminal.

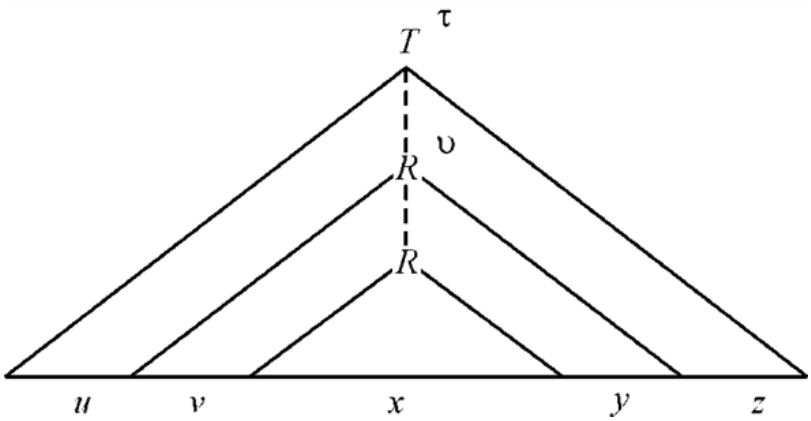
• There are only $|V|$ variables in the grammar $G$. The pigeonhole principle is applied with the variables (or non-terminal nodes) being the holes and with the pigeons being the variables encountered on any path in the tree.

• There are $|V|+1$ pigeons going into $|V|$ holes, so at least one hole must contain more than one pigeon. That is at least one variable occurs more once in the longest path of the smallest possible parse tree $\tau$.

Comment

**Step 4** of 8

Let this variable be $R$. Use the parse tree $\tau$ from the start symbol $T$ to divide the string $s$ into five pieces, of the form $uvxyz$. Consider the figure which is given below:

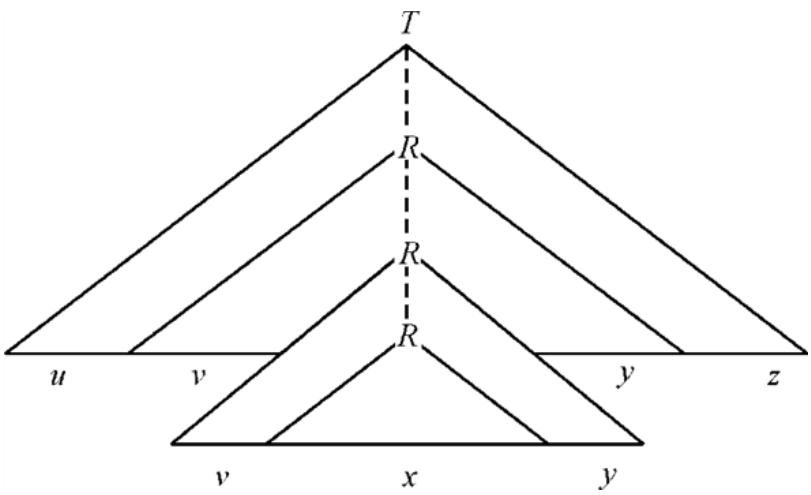



The parse tree $\upsilon$ at the first $R$ vertex hit by the start symbol $T$ breaks down the substring $vxy$ into three strings of terminals. Therefore this tree $\upsilon$ can be repeated any number of times. This is shown by considering a couple of examples.

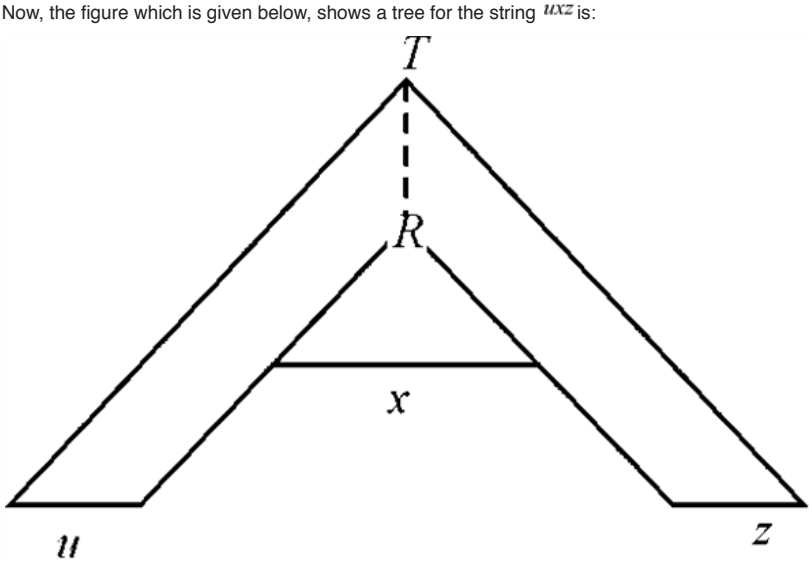
Comment

**Step 5** of 8

Consider the figure which is given below. It shows a tree for the string $uv^2xy^2z$ is:




Now, the figure which is given below, shows a tree for the string $uxz$ is:




**In this way, it can be said that "The first condition $uv^ixy^iz \in A, i \geq 0$ has been shown to be true".**

Comment

**Step 6** of 8

**In the second condition, it is** stated that **either of** $v$ **and** $y$ **cannot be the empty string** $\varepsilon$. The parse tree $\tau$ is the smallest possible tree.

• If $v = \varepsilon$ or if $y = \varepsilon$, then one or more vertices can be removed from the sub-trees ending in $\varepsilon$ in the parse tree $\tau$.

•

Comments (1)

**Step 7** of 8

This is a contradiction as it is not possible to remove vertices from the tree $\tau$ as it is the smallest possible parse tree.

**It has been shown that** $v \neq \varepsilon$ **and** $y \neq \varepsilon$ **.**

Comment

**Step 8** of 8

**Lastly it has to be proven that** $|vxy| \leq k$. Alternatively, the depth of the tree $\upsilon$ has to be at least $k$. Since the root node $R$ repeats and the path has $|V|+1$ variables. **Thus depth of the tree** $\upsilon$ **is at least** $|V|+1$ **and its yield will be at most** $b^{|V|+1} = k$**.**

Comment

## Problem

Refer to Problem 1.41 for the definition of the perfect shuffle operation. Show that the class of context-free languages is not closed under perfect shuffle.

Problem 1.41

For languages A and B, let the ***perfect shuffle*** of A and B be the language

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma\}.$$

Show that the class of regular languages is closed under perfect shuffle.

## Step-by-step solution

**Step 1** of 1

Context free language is the language which is generated by CFG or the context free grammar. It is possible to get different types of context free languages from different types of context free grammar.

Consider the languages A and B as given below:

Language *A* $\{0^k 1^k | k \geq 0\}$

Language *B* $\{a^k b^{3k} | k \geq 0\}$

After the perfect shuffle, other language (*C*) obtained will be $\{(0a)^k (0b)^k (1b)^{2k} \mid k >= 0\}$

It is clear that languages *A* and *B* are context free languages but language *C* is not a context-free language. It can be proved by contradiction follow:

• Suppose that the language defined for the shuffle of *A* and *B* is context free and let *p* be the length of its pumping lemma and *s* is string $(0a)^p (0b)^p (1b)^{2p}$.

• As string is longer as compared with pumping lemma and string is a part of language *C*, divide $s = uvxyz$.

• In language *C*, string is equal to one-fourth part of 1s and one-eighth part of *a*'s. For $uv^2xy^2z$, to have the same property, it should contain *a*'s and 1's. But it is not possible because both are separated by another symbol 2*p*. Also $|vxy| \leq p$.

Thus, it can be concluded and deduced that context free languages or CFL's are not closed under perfect shuffle.

Comment

## Problem

Refer to Problem 1.42 for the definition of the shuffle operation. Show that the class of context-free languages is not closed under shuffle.

Problem 1.42

For languages A and B, let the ***shuffle*** of A and B be the language

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma^*\}.$$

Show that the class of regular languages is closed under shuffle.

## Step-by-step solution

**Step 1** of 2

Context free language is the language which is generated by CFG or the context free grammar. It is possible to get different types of context free languages from different types of context free grammar.

Consider two languages as *P* and *Q* defined over the alphabet and let the shuffle operation defined for these two languages is:

$$\left\{ \mathrm{w} \mid \mathrm{w} = p_1 q_1 \ldots\ldots p_k q_k, \text{where } p_1 \ldots\ldots p_k \in P, q_1 \ldots\ldots q_k \in q \text{ and each } a_j b_j \in \Sigma^* \right\}$$

**Now, the two languages under consideration can be defined as follows:**

$$P = \left\{ w \in \{0,1\}^* \mid n_0(w) = n_1(w) \right\}$$

$$Q = \left\{ w \in \{p,q\}^* \mid n_p(w) = n_q(w) \right\}$$

Where, in the above grammar $n_1(w)$ is the number of one's in *w* and $n_0(w)$ is the number of zeros present in the string *w*. Also, $n_p(w)$ and $n_q(w)$ denotes the number of *p's* and number of *q's* in *w* respectively.

Comment

**Step 2** of 2

**The language belonging to the shuffle of *P* and *Q* can be given as:**

$$S = \left\{ w \in \{0,1,p,q\}^* \mid n_0(w) = n_1(w) \text{ and } n_p(w) = n_q(w) \right\}$$

**The language shown above for the shuffle of *P* and *Q* is not a context free language and it can be proved by using the proof by contradiction**.

Therefore, suppose that the language defined for the shuffle of P and Q is context free and let *t* be the length of its pumping lemma.

• Now, also suppose that $ab^2de^2f \neq S$ and by using the approach of pumping lemma shuffle can be represented or written as *abdef* where $|ae| > 1$ and $|\mathrm{abd}| \leq r$. Now, $|\mathrm{abd}|$ cannot contain both zeros and ones and both *p's* and *q's* as $|\mathrm{abd}| \leq r$.

• Thus, if the principle of pumping lemma is applied to $ab^2de^2f$ once then the resultant string that will be obtained will have unequal number of zeros and ones or unbalanced ratio of *p's* and *q's*.

• This implies that $ab^2de^2f \neq S$ and thus this also signifies that the language S belonging to the shuffle of *P* and *Q* is not a context free language or CFL.

Now, the two languages *P* and *Q* which are under consideration are context free but the shuffle of these two languages is not context free.

**Thus, it can be concluded and deduced that context free languages or CFL's are not closed under shuffle operation**.

Comments (2)

## Problem

Say that a language is ***prefix-closed*** if all prefixes of every string in the language are also in the language. Let C be an infinite, prefix-closed, context-free language. Show that C contains an infinite regular subset.

## Step-by-step solution

**Step 1** of 2

**Given:** A language which is prefix closed. Here, it is required to prove that an infinite regular subset is contained in every context free language which is prefix-closed.

Comment

**Step 2** of 2

**Proof:** Consider a language L which is context free and prefix closed. As the language L is context free and therefore, the principle of pumping lemma can be applied on it.

Let the length of the pumping lemma be P and the string that needs to be considered in the language is s.

The length of the pumping lemma P is lesser or smaller than length of the string. Now, it is possible to break the string into 'abcde' such that $ab^k cd^k e \in L$ for all $k \geq 0$ and $|bd| \geq 1$.

Now, it is already given that the language L is prefix closed and all the prefixes of the string s are also present in language L, therefore, $ab^k \in L$ for all $k \geq 0$.

Thus, it can be inferred that the language formed from the $ab^*$ is regular and it is a subset of L.

**Also, if $b \neq \in$ then it implies $ab^*$ is an infinite regular subset of Land thus it proves the required statement**.

Comments (2)

## Problem

Read the definitions of NOPREFIX(A) and NOEXTEND(A) in Problem 1.40.

**a.** Show that the class of CFLs is not closed under NOPREFIX.

**b.** Show that the class of CFLs is not closed under NOEXTEND.

Problem 1.40

Recall that string x is a ***prefix*** of string y if a string z exists where xz = y, and that x is a ***proper prefix*** of y if in addition x ≠ y. In each of the following parts, we define an operation on a language A. Show that the class of regular languages is closed under that operation.

A**a.** NOPREFIX(A) = {w ? Al no proper prefix of w is a member of A}.

**b.** NOEXTEND(A) = {w ? Alw is not the proper prefix of any string in A}.

## Step-by-step solution

**Step 1** of 3

a)

Consider the $NOPREFIX$ operation. For a language $A$, the $NOPREFIX$ operation is defined as:

$NOPREFIX(A) = \{w \in A \mid \text{no proper prefix of } w \text{ is a member of } A\}$

• Now consider the language $P$, defined as $P = P_1 \cup P_2$ where $P_1 = \{x^a y^b z \mid a \neq b, a, b \geq 1\}$ and $P_2 = \{x^a y^b z^b \mid a, b \geq 1\}$.

• If a string $x^a y^b z$ of $P_1$ is considered, then the proper prefix of it is the string that consists only $x$ and $y$ and all the string in $P_1$ and $P_2$ consists minimum one $z$. Therefore, all strings in $P_1$ is in $NOPREFIX(P)$.

Comment

**Step 2** of 3

Now, if a string $x^a y^b z^b$ in $P_2$ is considered. It is not in $NOPREFIX(P)$, if and only if there is proper prefix of it that is in $P$.

• As no proper prefix exists in $P_2$, the proper prefix will have to come from $P_1$ and hence the $a \neq b$. Thus, the string in $P_2$ which are in $NOPREFIX(P)$ are $\{x^a y^a z^a \mid a \geq 1\}$. Therefore, $NOPREFIX(P) = P_1 \cup \{x^a y^a z^a \mid a \geq 1\}$

• $P$ is a context free language since $P_1$ and $P_2$ are both context-free and context-free languages are closed under union. However, $NOPREFIX(P)$ is not context-free.

• In other way, context-free behavior is shown by $NOPREFIX(P) \cap P(x^* y^* zzz^*) = \{x^a y^a z^a \mid a \geq 2\}$ **that is a contradiction**. Therefore, **a context-free language** $P$ **exists in such a way that** $NOPREFIX(P)$ **is not context-free.**

Hence, from the above discussion, it can be said that **context-free languages are not closed under** $NOPREFIX$ **operation.**

Comment

**Step 3** of 3

b)

Consider the $NOEXTEND$ operation. For a language $A$, the $NOEXTEND$ operation is defined as:

$NOEXTEND(P) = \{w \in A \mid w \text{ is not a proper prefix of any string in } A\}$

Now consider the language $P = P_1 \cup P_2$ where $P_1 = \{x^a y^b z^c \mid a \neq b, a, b, c \geq 1\}$ and $P_2 = \{x^a y^b z^b \mid a, b \geq 1\}$.

• Consider the string $x^a y^b z^c \in P_1$, the given string is not in $NOEXTEND(P)$ since $x^a y^b z^{c+1}$, which is an extension of the string is in $P$.

• Now, the string $x^a y^b z^b$ is considered. Now any extension of this string in $P$ should belong to $P_1$. Hence this string will not exist in $NOEXTEND(P)$, if and only if an extension of it belongs to $P_1$ if $a \neq b$. Therefore**, the string of the form** $x^a y^a z^a$ **belongs to** $NOEXTEND(P)$**.**

• Hence, $NOEXTEND(A) = \{x^a y^a z^a \mid a \geq 1\}$. As it is known **that** $P$ **is context-free but** $NOEXTEND(P)$ **is not context-free**.

Hence from the above explanation, it can be said that "**the context-free language are not closed under** $NOEXTEND$ **operation".**

Comment

## Problem

Let

$$Y = \{w \mid w = t_1\#t_2\#\cdots\#t_k \text{ for } k \geq 0, \text{ each } t_i \in 1^*, \text{ and } t_i \neq t_j \text{ whenever } i \neq j\}.$$

Here ∑ = {1, #}. Prove that Y is not context free.

## Step-by-step solution

**Step 1** of 6

**CFG:**

• A fixed set of grammar rules is known as **CFG (context free grammar)**. It consisting of that is augment (N, T, P, S).

• Where, **N** is set of non-terminal symbol.

• **T** is set of terminal $N \bigcap T = NULL$.

• **P** is set of rule, $P : N \to (N \bigcup T)^*$.

• **S** is start symbol.

Comment

**Step 2** of 6

**Consider the following details:**

The language is $Y = \{w \mid w = t_1\#t_2\#....\#t_k \text{ where } k \geq 0, t_i \in 1^* \text{ and } t_i \neq t_j \text{ when } i \neq j\}$ with the terminals being $\Sigma = \{1, \#\}$.

Comment

**Step 3** of 6

**Proof:**

Theorem 2.34: Any string $s$ in $A$, the pumping lemma $p$ is the minimum length such that It could make part under five ends $s = uvxyz$. The string $s$ also satisfies the following conditions for a context-free language $A$:

1. The string what's to come for $uv^ixy^iz$ only those context-free dialect $A$, the point when every $i \geq 0$.

$uv^ixy^iz \in A$

2. Those strings that need aid pumped, $v$ furthermore $y$, can't both make the void string $\varepsilon$.

$|vy| > 0$

3. The joined together period of the strings lying inside what's to come for $u$ Also $z$ must not a chance to be more stupendous that those pumping length $p$.

$|vxy| \leq p$

Comment

**Step 4** of 6

**This problem is solved by the proof of contradiction.**

• The language $Y$ is supposed to be a context-free language.

• Theorem 2.34 is shown not to hold for the language.

• This makes the assumption, which is that $Y$ is a CFL, invalid.

• Assume language $Y$ is a context-free language.

• Now it can be seen that either $x$ or $y$ cannot have any $\#$'s.

• This is as when user pump the string then user will get strings of the form $s = t_1 \# t_2 \# .... \# t_k$ where $t_i = t_j$ when $i \neq j$.

• Such strings do not lie in $A$. Consequently, to get $v, y = 1^*$.

Comment

**Step 5** of 6

**Construction:**

Consider the string $s = uv^i xy^i z$ with $v = 1^*, y = 1^*$. The two cases possible for the substring $vxy$ are:

• The substring contains $\#$: as the $\#$ symbol cannot lie in either $v$ or $y$, it must lie in $x$.

• When the string $s = uv^i xy^i z$ is pumped with $x = 1^* \# 1^*$, the case $t_i = t_j$ when $i \neq j$ can occur.

• Thus, pumping this string does not necessarily produce strings that lie in $Y$.

• It does not contain $\#$: the substring $xyz$ will be just be a sequence of $1$s. As was argued for the previous case, on applying the condition 1 of theorem 2.34 to pump the string will result in strings wherein $t_i = t_j$ in cases when $i \neq j$. As has been seen these strings are not part of language $Y$.

Comments (1)

**Step 6** of 6

**Conclusion:**

The language $Y$ does not satisfy the pumping lemma. Consequently, it is not a CFL.

Comment

## Problem

Next

For strings w and t, write $w \doteq t$ if t and w have the same symbols in the same quantities, but possibly in a different order.

For any string $w$, define $SCRAMBLE(w) = \{t \mid t \doteq w\}$. For any language $A$, let $SCRAMBLE(A) = \{t \mid t \in SCRAMBLE(w) \text{ for some } w \in A\}$.

**a.** Show that if ∑ = {0,1}, then the *SCRAMBLE* of a regular language is context free.

**b.** What happens in part (a) if ∑ contains three or more symbols? Prove your answer.

## Step-by-step solution

**Step 1** of 4

For any two strings *w* and *t*, *SCRAMBLE* is defined as, $SCRAMBLE(w) = \{t \mid t \square w\}$. Consider a language A, $SCRAMBLE(A) = \{t \mid t \in SCRAMBLE(w) \text{ for some } w \in A\}$.

Comment

**Step 2** of 4

a.

Consider the input alphabet $\Sigma = \{0,1\}$. Assume the language $L = (10)^*$ over the input alphabet $\Sigma = \{0,1\}$. The *SCRAMBLE*(*w*) can have all the word permutations of *w*. The states in FA for the language *L* must be rearranged to accept the *SCRAMBLE* of the language.

The intersection of *SCRAMBLE*(*L*) and a regular language need to be regular if *SCRAMBLE*(*L*) is regular. Consider a regular language $M = 1^*0^*$. The intersection of *SCRAMBLE*(*L*) and *M* is, $SCRAMBLE(L) \cap 1^*0^* = \{1^n0^n : n \geq 0\}$ which is context free language. Thus, the *SCRAMBLE*(*L*) cannot be a regular language.

Comment

**Step 3** of 4

Now, using push down automata *P* to show that scramble $SCRAMBLE(A)$ of regular language *A* is context free. The Push down automata *P* contains set of states similar to DFA *D* which accepts *A*. Consider the input $w = w_1w_2w_3 \cdots w_n$, the push down automata *P* non-deterministically works to guess the permutation $u$ of $w$ in $A$ regular language. When it guesses $w_1 = u_1$ then transition occurs by reading the input $w_1$ like *D*. When it guesses $w_1 \neq u_1$ then null transition occurs. The input symbol $u_1$ stored in stack for the future use.

When number of elements in the stack is greater than 1 then the machine works as follows:

If the input symbol is same as the top element of the stack, then it consumes both the symbol but does not make any transition. If the input symbol does not matches with the top element of the stack then it guesses that may be it is the next right element in *u* or guesses that current input symbol is wrong and make null transition while push the complement of the symbol in the stack. The symbol that is inserted in the stack will pay back as soon as possible. For language *A,* if *w* is a permutation for string *u* then there must be a path that lead to the accept state with empty state. This will be the acceptance conditions for PDA *P*.

**Therefore, the SCRAMBLE of a regular language is context free.**

Comment

**Step 4** of 4

b.

Consider the input alphabet $\Sigma = \{0,1,2\}$. Assume the language $L = (210)^*$ over the input alphabet $\Sigma = \{0,1,2\}$.

The intersection of *SCRAMBLE*(*L*) and a regular language need to be regular if *SCRAMBLE*(*L*) is regular. Consider a regular language $M = 2^*1^*0^*$.

The intersection of *SCRAMBLE*(*L*) and *M* is, $SCRAMBLE(L) \cap 2^*1^*0^* = \{2^n1^n0^n : n \geq 0\}$ which is not a ntext free language.

Next

The pushdown automata has a single stack which can be used to compare the occurrences of two symbols, but for 3 or more symbols PDA cannot be drawn.

**Therefore, the SCRAMBLE of a regular language with 3 or more symbols is not context free.**

Comment

## Problem

If A and B are languages, define $A \diamond B = \{xy \mid x \in A \text{ and } y \in B \text{ and } |x| = |y|\}.$

## Step-by-step solution

**Step 1** of 3

Consider the two regular languages *A* and *B* over the input alphabet $\Sigma$. The language $A \diamond B$ is defined as, $A \diamond B = \{xy \mid x \in A \text{ and } y \in B \text{ and } |x| = |y|\}$. For the language $A \diamond B$, if PDA is constructed then it can be said that $A \diamond B$ is in CFL.

Comment

**Step 2** of 3

Consider the DFA $D_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ and $D_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ for the languages *A* and *B* respectively. The strings of the language $A \diamond B$ are formed by concatenating equal length strings from *A* and *B*. Construct the PDA $M = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$ for the language $A \diamond B$ as,

• From the start state $q_{start}$, push the symbol $ into the stack to know the bottom of the stack.

• For every symbol from the language *A*, push 1 into the stack. It guesses the end of the string that belongs to *A* when it reaches to the final state of $D_A$.

• Then, for every symbol from the language *B*, pop 1 from the stack. When the string reaches the final state of $D_B$, it moves to the final state *F* if and only if the top of the stack is $.

The PDA can be constructed informally as above described.

Comment

**Step 3** of 3

The formal description of the PDA $M = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$ is as follows:

- $Q = Q_A \cup Q_B \cup \{q_{start}, F\}$
- $\Sigma$ is the input alphabet for *A* and *B*
- $\Gamma = \{\$, 1\}$ where $ is the symbol used to know the bottom of the stack and 1 is pushed every time into the stack when the symbol read from the language *A*.

• The transition function is defined as,

$$\begin{aligned}
&\delta(q_{start}, \varepsilon, \varepsilon) = \{q_A, \$\} \\
&\delta(q, a, \varepsilon) = \{\delta_A(q, a), 1\} && \text{if } q \in Q_A, \quad a \in \Sigma \\
&\delta(q, \varepsilon, \varepsilon) = \{q_B, \varepsilon\} && \text{if } q \in F_A \\
&\delta(q, a, 1) = \{\delta_B(q, a), \varepsilon\} && \text{if } q \in Q_B, \quad a \in \Sigma \\
&\delta(q, \varepsilon, \$) = \{F, \varepsilon\} && \text{if } q \in F_B
\end{aligned}$$

Any other transitions apart from this will not be accepted.

- $q_{start}$ is the start state.
- *F* is the final state.

The PDA non deterministically guesses the end of the string from $D_A$ and transitions to the start symbol of $D_B$ if it is in a final state of $D_A$. The PDA *M* accepts the string when it is in accepting state of $D_B$ while hitting the empty stack. This shows that $L(M) = A \diamond B$.

Therefore, for any two regular languages *A* and *B*, $A \diamond B$ is in CFL.

## Problem

Let $A = \{wtw^{\mathcal{R}} \mid w, t \in \{0,1\}^* \text{ and } |w| = |t|\}.$ Prove that A is not a CFL.

## Step-by-step solution

**Step 1** of 2

In the given function $A = \left\{ wtw^R \middle| w, t \in \{0,1\}^* \text{ and } |w| = |t| \right\}$, every string $s = wtw^R \in A$ where $|w| = |t| = |w^R|$ and $|s|$ is a multiple of three. Let us assume that *A* is context free and reach to a contradiction.

• Let *p* be the pumping length for A that is guaranteed to exit by pumping lemma.

• Select string $s = 0^{2p}0^p1^p0^{2p} \in A$ with $|s| > p$.

• Therefore, there exits $uvxyz$ such that

1) $uv^ixy^iz \in B$ for all $i \geq 0,$

2) $|uy| > 0$,

3) $|vxy| \leq p$.

• Consider these cases for pumping lemma:

**Case 1**: $|vy|$ is not a multiple of 3. Then $s' = uv^2xy^2z \notin A$ since $|s'|$ is no longer a multiple of 3.

**Case 2**: $vxy$ consist of only 0s from the prefix set of 0s and $|vy| = 3r$ for some *r*. Then, $uv^2xy^2z = 0^{3p+3r}1^p0^{2p} = 0^{2p+r}0^{p+2r}1^{p-r}1^r0^{2p} \notin A$, since $w = 0^{2p+r}$ and $w^R \neq 1^r0^{2p}$, the $w^R$ of the string $s$.

**Case 3**: $uxy$ consists of only 1s and $|vy| = 3r$ for some *r*. Then, the string $uv^2xy^2z = 0^{3p}1^{p+3r}0^{2p} = 0^{2p+r}0^{p-r}1^{p+2r}1^r0^{2p} \notin A$, since $w = 0^{2p+r}$ and $w^R \neq 1^r0^{2p}$, the $w^R$ of the string $s$.

**Case 4**: $vxy$ consist of only 0s from the suffix set of 0s and $|vy| = 3r$ from some *r*. Then $uv^0xy^0z = 0^{3p}1^p0^{2p-2r} \notin A,$ since $w = 0^{2p-r}$ and $w^R \neq 1^{2r}0^{2p-3r}$, the $w^R$ of the string $s$.

.

**Case 5**: $uy = 0^m1^n$ with $m, n > 0$ and $m + n = 3r$ from some *r*. Then, the string $uv^2xy^2z = 0^{3p+m}1^{p+n}0^{2p} = 0^{2p+r}0^{p+m-r}1^{p+n-r}1^r0^{2p} \notin A,$ since $w = 0^{2p+r}$ and $w^R \neq 1^r0^{2p}$, the $w^R$ of the string $s$.

**Case 6**: $vy = 1^m0^n$ with $m, n > 0$ and $m + n = 3r$ for some *r*.

• **Sub-case 6.1**: $n < r$. Then $uv^2xy^2z = 0^{3p}1^{p+m}0^{2p+n} = 0^{2p+r}0^{p-r}1^{p+m+n-r}1^{r-n}0^{2p+n} \notin A$, since $w = 0^{2p+r}$ and $w^R \neq 1^{r-n}0^{2p+n}$, the $w^R$ of the string $s$.

• **Sub-case 6.2**: $n > r$. Then $uv^0xy^0z = 0^{3p}1^{p-m}0^{2p-n} = 0^{2p-r}0^{p+r}1^{p+r-m-n}1^{n-r}0^{2p-n} \notin A$, since $w = 0^{2p-r}$ and $w^R \neq 1^{n-r}0^{2p-n}$, the $w^R$ of the string $s$.

• **Sub-case 6.3**: $n = r$. Then $uv^{p+2}xy^{p+2}z = 0^{3p}1^{p+2r(p+2-1)}0^{2p+r(p+2-1)n} = 0^{3p}1^{rp+r-p}1^{2r+rp+r}0^{2p+rp+r} \notin A$, since $w = 0^{2p+r}$ and $w \neq 0^{3p}1^{rp+r-p}$ and $w^R \neq 0^{2p+rp+r}$, the $w^R$ of the string $s$.

If $i < p+2,$ then take $r = 1$, $uv^ixy^iz = 0^{3p}1^{p+2(i-1)}0^{2p+(i-1)}$, $= 0^{2p+(i-1)}0^{p-(i-1)}1^{p+2(i-1)}0^{2p+(i-1)} \in A$. As there are not enough 1's to push or pump into *w,* $p+2$ is the first time that is guaranteed.

Comment

**Step 2** of 2

**Thus, in all the cases, the 1) of pumping lemma results in a contradiction. Therefore, the assumption that *A* is context free language, is false.**

Comment

## Problem

Consider the following CFG G:

$$S \rightarrow SS \mid T$$
$$T \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{ab}$$

Describe L(G) and show that G is ambiguous. Give an unambiguous grammar H where L(H) = L(G) and sketch a proof that H is unambiguous.

## Step-by-step solution

**Step 1** of 10

**CFG:**

• A Context Free Grammar (CFG) is an arrangement of recursive rewriting principles (or productions) used to create pattern of strings.

• A CFG comprises of the following components: An arrangement of terminal symbols.

• Which are the characters of the letter set that show up in the strings created by the grammar.

Comment

**Step 2** of 10

**Ambiguous Grammar:**

• This is a context free grammar to which there exists a string that can have in excess of left most derivation or parse tree.

Comment

**Step 3** of 10

**The language of a CFG $G$ has to be described and shown to be ambiguous.**

**An unambiguous grammar $H$ has to obtain from the grammar.**

**The rules for the CFG $G$ are:**

• First, consider the strings produced by the variable $T$. Let the CFG $I = (V, \Sigma, R, T)$ be:

$T \rightarrow aTb \mid ab$

• As it is either a string of two terminals $ab$ or it is placed between the same two terminals $aTb$.

• Therefore, the language generated will consist of a sequence of two or more $a$'s followed by the same number of $b$'s.

$L(I) = \{a^i b^i \mid i > 1\}$

• All the strings lying in this language will be of even length as $\left|a^i b^i\right| = i + i = 2i$.

Comment

**Step 4** of 10

• The start symbol $S$ either can be replaced by two start symbols $SS$ or by the variable $T$.

• It can be seen that a string in the language $L(G)$ will be the concatenation of one or more occurrences of strings of $L(I)$. So, the language $L(G)$ will be given by:

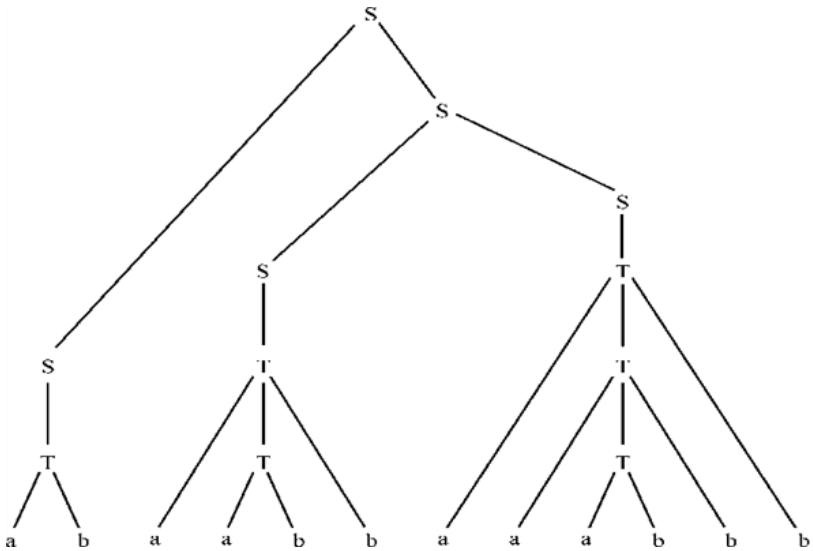
$$L(G) = \{a^{i_1}b^{i_1}a^{i_2}b^{i_2}....a^{i_k}b^{i_k} \mid i_1, i_2, ...., i_k \geq 1 \, and \, k \geq 1\}$$

• Take the string $s = abaabbaaabbb$.

• A derivation for this string $s$ is:

$$S \Rightarrow SS \Rightarrow TS \Rightarrow abS \Rightarrow abSS \Rightarrow abTS \Rightarrow abaTbS \Rightarrow abaabbS$$
$$\Rightarrow abaabbT \Rightarrow abaabbaTb \Rightarrow abaabbaaTbb \Rightarrow abaabbaaabbb$$

• This leads to parse tree:



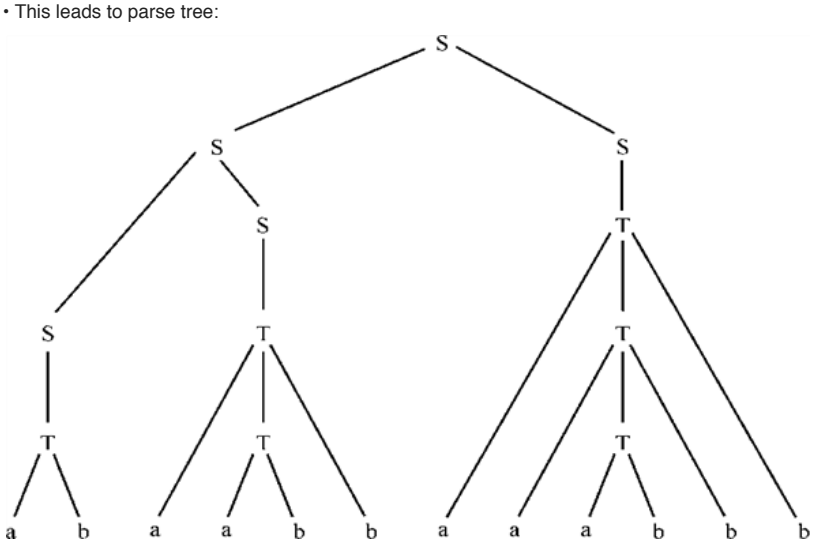

Comment

**Step 5** of 10

**The Second alternative derivation for this string $s$ is:**

$$S \Rightarrow SS \Rightarrow ST \Rightarrow SaTb \Rightarrow SaaTbb \Rightarrow Saaabbb \Rightarrow SSaaabbb \Rightarrow STaaabbb$$
$$\Rightarrow SaTbaaabbb \Rightarrow Saabbaaabbb \Rightarrow Taabbaaabbb \Rightarrow abaabbaaabbb$$

• This leads to parse tree:




• The string $s = abaabbaaabbb$ can be derived into two different parse trees. So, the grammar $G$ is ambiguous. The rules for the start symbol $S$ are:

$$S \rightarrow SS \mid T$$

• Due to this rule, it is possible to get different derivations for a string in the language $L(G)$.

Comment

**Step 6** of 10

**Unambiguous Grammar:**

• This is also a context free grammar to which each substantial string has a one of a kind unique left most derivation or parse tree.

• The grammar $G$ can be converted into unambiguous grammar $H = (V, \Sigma, R', S)$ by removing this ambiguity. The rules are modified to:

$S \to TS \mid T$
$T \to aTb \mid ab$

• It has to be checked if $L(G) = L(H)$.

• That is a string $s$ lies in $L(G)$ if and only if it lies in $L(H)$. This result is proven via induction on the length of $s$.

•

Comment

**Step 7** of 10

It has to be proven in both the directions – the 'if' direction and the 'only if' direction.

Comment

**Step 8** of 10

**Now, it can be proved that a string $s$ lies in $L(G)$ if it lies in $L(H)$.**

**Basis**:

• The string $ab$ lies in $L(H)$.

• It also be derived in $L(G)$, that is $S \Rightarrow_G T \Rightarrow_G ab$

**Inductive step**:

• All strings of length at most $n$ of $L(H)$ lie in the language $L(G)$. Consider an arbitrary $w$ string of length $n+2$ lying in the language $L(H)$.

• There are three cases possible:

1. The string $w$ starts with an $ab$. It can be derived in the language $L(G)$ to a string of length $n$ as follows.

$S \Rightarrow_G SS \Rightarrow_G TS \Rightarrow_G abS \Rightarrow_G abx$

2. An $ab$ in the middle of the string $w$. The derivation for this string is.

$S \Rightarrow_G SS \Rightarrow_G SSS \Rightarrow_G STS \Rightarrow_G SaTbS \Rightarrow_G xaybz$

3. The string $w$ ends with an $\mathrm{ab}$.

$S \Rightarrow_G SS \Rightarrow_G ST \Rightarrow_G Sab \Rightarrow_G xab$

• In all the above cases, the string can be derived in grammar $G$ to a string(s) whose length $\le n$. So, a string $s$ lies in $L(G)$ if it lies in $L(H)$.

Comment

**Step 9** of 10

**Only-If**:

• It can be proved that a string $s$ lies in $L(H)$ if it lies in $L(G)$.

**Basis**:

• The string $ab$ lies in $L(G)$. It also be derived in $L(H)$, that is $S \Rightarrow_H T \Rightarrow_H ab$.

**Inductive step**:

• All strings whose length is equal to or less than $n$ of $L(G)$ lie in the language $L(H)$

Comment

**Consider an arbitrary $w$ string of length $n+2$ lying in the language $L(G)$ .**

• The three cases possible are:

• The string $w$ starts with an $ab$. It can be derived in the language $L(H)$ to a string of length $n$ as follows.

$$S \Rightarrow_H SS \Rightarrow_H TS \Rightarrow_H abS \Rightarrow_H abx$$

• An $ab$ in the middle of the string $w$. The derivation for this string is.

$$S \Rightarrow_H SS \Rightarrow_H SSS \Rightarrow_H STS \Rightarrow_H SaTbS \Rightarrow_H xaybz$$

• The string $w$ ends with an $ab$.

$$S \Rightarrow_H SS \Rightarrow_H ST \Rightarrow_H Sab \Rightarrow_H xab$$

The string can be derived in all the cases into a string(s) whose length is at most.

• Therefore, it has been proven if a string lies in $L(H)$ then it also lies in $L(G)$.

• An outline of a proof showing that $H$ **is unambiguous is to be given**.

• A grammar is unambiguous if there is only one leftmost derivation of a string in the grammar.

• An induction on the length of the string can be used to prove this and showing that is only one way to derive an arbitrary string into a string of smaller length.

**Hence, it has been proven that the languages $L(H)$ and $L(G)$ are equivalent. In other words:**

$$L(G) = L(H)$$

Comment

## Problem

Let ∑ = {0,1} and let B be the collection of strings that contain at least one 1 in their second half. In other words,

$$B = \{uv \mid u \in \Sigma^*, v \in \Sigma^* 1 \Sigma^* \text{ and } |u| \geq |v|\}.$$

**a.** Give a PDA that recognizes B.

**b.** Give a CFG that generates B.

## Step-by-step solution

**Step 1** of 5

Consider the following information:

The alphabet set $\Sigma = \{0,1\}$.

$B = \{uv \mid u \in \Sigma^*, v \in \Sigma^* 1 \Sigma^* \; and \; |u| \geq |v|\}$.

Comment

**Step 2** of 5

**a.**

**Construction of PDA:**

1. The PDA for this language works by first reading and pushing the string u onto the stack. The string u consists of either 0's or 1's.

2. After pushing the string *u* into stack is completed, the machine has all the letters of *u* on the stack. Now, match the letters in *u* with the letters in *v*.

3. Remember that the length of *u* is greater than or equal to *v*.

4. If the condition that number of 1's in the input is at least 1 in second half, then accept the entire string.

**Problem:**

• But here a problem takes place in finding the middle of the string *uv*.

• That is the position at which *u* ends and *v* starts.

• The problem can be solved by using **Non deterministic push down automata**.

• The PDA will be designed such that the empty string $ can be accepted by the PDA.

**Construction:**

The PDA $M$ that recognizes $B$ is $(Q, \Sigma \frac{1}{2}, \Gamma, \delta, q_0, F)$, where:

$Q = \{q_0, q_1, q_2, q_3\}$,
$\Sigma = \{0,1\}$,
$\Gamma = \{x\}$,

$F = \{q_3\}$

Transition function $\delta$ of the represented in a tabular format:

| *Input* : | 0 | | 1 | | $\varepsilon$ | |
|---|---|---|---|---|---|---|
| *Stack* : | $x$ | $\varepsilon$ | $x$ | $\varepsilon$ | $x$ | $\varepsilon$ |
| $q_0$ | $\{(q_0, xx)\}$ | $\{(q_0, x)\}$ | $\{(q_0, xx)\}$ | $\{(q_0, x)\}$ | $\{(q_1, x)\}$ | |
| $q_1$ | $\{(q_1, \varepsilon)\}$ | | $\{(q_2, \varepsilon)\}$ | | | |
| $q_2$ | $\{(q_2, \varepsilon)\}$ | | $\{(q_2, \varepsilon)\}$ | | | |

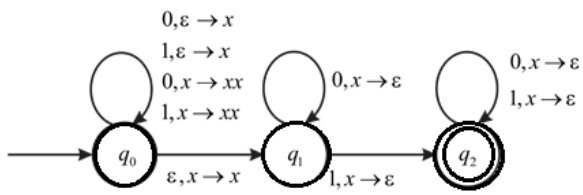
Comment

**Step 3** of 5

The state diagram for the PDA $M$ is given below:

Comment

**Step 4** of 5




Comments (6)

**Step 5** of 5

**b.**

The CFG for the language *B* is as follows:

$$S \to UV$$
$$U \to AB$$
$$V \to A1A \mid A1B \mid AIU \mid B1U \mid U1U$$
$$A \to 00^* \mid \in$$
$$B \to 11^* \mid \in$$

**Explanation:**

• The string *S* is concatenation of *U* and *V*.

• The string U may consist of any number of 0's and 1's.

• The string *V* may consist of atleast one 1 or only one 1.

• The string *A* may consists of atleast one zero or more than one zeros.

• The string *B* may consist of atleast one 1 or more than one 1's.

Comments (3)

## Problem

Let $\Sigma = \{0,1\}$. Let $C_1$ be the language of all strings that contain a 1 in their middle third. Let $C_2$ be the language of all strings that contain two 1s in their middle third.

$$\text{So } C_1 = \{xyz \mid x, z \in \Sigma^* \text{ and } y \in \Sigma^* 1 \Sigma^*, \text{ where } |x| = |z| \geq |y|\}$$

$$\text{and } C_2 = \{xyz \mid x, z \in \Sigma^* \text{ and } y \in \Sigma^* 1 \Sigma^* 1 \Sigma^*, \text{ where } |x| = |z| \geq |y|\}.$$

**a.** Show that $C_1$ is a CFL.

**b.** Show that $C_2$ is not a CFL.

## Step-by-step solution

**Step 1** of 7

**A language** $L$ is said to be **context-free** if there exist some integer $q \geq 1$ (it is also known as pumping length) in such a way that all the string $S$ in $L$ which is equal or longer than $q$ symbols or $|S| \geq q$.

It can be written as $S = abcde$ with substring $a, b, c, d \text{ and } e$ such that

1. $|bcd| \leq q$
2. $|bd| \geq 1$, and
3. $ab^x cd^x e$ is in $L$ for all $x \geq 0$.

Comment

**Step 2** of 7

**a.**

Consider the language $C_1$ which is given below:

$C_1 = \{xyz \mid x, z \in \Sigma^* \text{ and } y \in \Sigma^* 1 \Sigma^*, \text{where} |x| = |z| \geq y\}$, where $\Sigma = \{0,1\}$

Now, a Push down automata needs to construct which help in determining the language $C_1$. The PDA $M$ that recognizes $C_1$ is $(Q, \Sigma \frac{1}{3}, \Gamma, \delta, q_0, F)$, where:

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0,1\}$
$\Gamma = \{x\}$
$F = \{q_2\}$

Comment

**Step 3** of 7

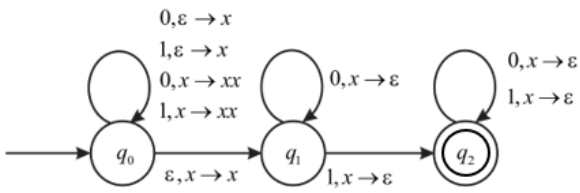
The transition functions $\delta$ of the represented in a tabular format:

| *Input* : | 0 | | 1 | | $\varepsilon$ | |
|---|---|---|---|---|---|---|
| *Stack* : | $x$ | $\varepsilon$ | $x$ | $\varepsilon$ | $x$ | $\varepsilon$ |
| $q_0$ | $\{(q_0, xx)\}$ | $\{(q_0, x)\}$ | $\{(q_0, xx)\}$ | $\{(q_0, x)\}$ | $\{(q_1, x)\}$ | |
| $q_1$ | $\{(q_1, \varepsilon)\}$ | | $\{(q_2, \varepsilon)\}$ | | | |
| $q_2$ | $\{(q_2, \varepsilon)\}$ | | $\{(q_2, \varepsilon)\}$ | | | |

Comment

**Step 4** of 7

The state diagram for the PDA $M$ is given below:




Hence from the above explanation it can be said that there exists a Turing machine which accepts the given language.

**Hence, $C_1$ is a context free language**.

Comments (4)

**Step 5** of 7

**b.**

Now consider the language $C_2$ that accepts all the string that contains two 1's in their middle of the string. The language $C_2$ can be defined as:

$C_2 = \{xyz \mid x, z \in \Sigma^* \text{ and } y \in \Sigma^* 1 \Sigma^* 1 \Sigma^*, \text{where} \mid x \mid = \mid z \mid \geq y\}$

Now, using a pumping lemma to show Language $C_2$ is not CFL.

Let as assume that $C_2$ is CFL and obtain a contradiction.

Let pumping length of the pumping lemma is *p*.

Let select a string $S = 0^{p+2}10^p10^{p+2}$ of given language.

Comment

**Step 6** of 7

Let, divide S into five pieces $S = uvxyz$, it must satisfy the conditions according to the pumping lemma,

1. For each $i \geq 0, uv^ixy^iz \in C_2$,
2. $\mid vy \mid > 0$, and
3. $\mid vxy \mid \leq p$

Where

$u = 0^{p+2}$

$v = 1$

$x = 0^p$

$y = 1$

$z = 0^{p+2}$

Comments (1)

**Step 7** of 7

Let $i = 1$

After pumping, string becomes $S = 0^{p+2}10^p10^{p+2}$.

According to the pumping lemma third condition, new string $\mid 10^p1 \mid \leq p$ becomes fails, that means, length of the *vxz* is greater than the pumping length *p*, i.e. $\mid vxy \mid \leq p$.

**Hence, $C_2$ is not context free language.**

## Problem

We defined the rotational closure of language A to be *RC*(A) = {yxl xy ? A}. Show that the class of CFLs is closed under rotational closure.

## Step-by-step solution

**Step 1** of 1

**CFLs is closed under rotational closure**

The rotational closure of a language $A$ is defined as $RC(A) = \{yx \mid xy \in A\}$.

The class of CFLs is closed under the concatenation operation. In other words if $L_1$ and $L_2$ are CFLs then the language $L = L_1 \bullet L_2$ is also a CFL. The converse is also true: if language $L = L_1 \bullet L_2$ is a CFL, then language $L_1$ and language $L_2$ are CFLs.

Consider a string $s = xy$ in the language $A$. It can be formed from the concatenation of two languages $X$ and $Y$, such that $x \in X$ and $y \in Y$. As the language $A = X \bullet Y$ is a CFL, the languages $X$ and $Y$ will also be CFLs.

The rotational closure of the string $s = xy$ will be $RC(s) = RC(xy) = yx$. It can be formed by concatenating $Y$ and $X$.

$RC(s) = Y \bullet X$

As both $Y$ and $X$ are context-free languages, the language $RC(s)$ is a context-free language for any string $s \in A$.

The rotational closure has been proven for the class of CFLs.

Comments (1)

## Problem

We defined the *CUT* of language A to be *CUT*(A) = {yxzl xyz ? A}. Show that the class of CFLs is not closed under *CUT*.

## Step-by-step solution

**Step 1** of 4

A CFL language is closed under few operations. If $L_1$ and $L_2$ are two CFL languages then the following language will also be context free:

1. Cyclic shift of CFL
2. Union of both is CFL
3. The reversal of $L_1$
4. Concatenation of both languages
5. Kleen star of $L_1$
6. Image of $L_1$ under homomorphism
7. Image inverse of $L_1$ under inverse homomorphism

Comment

**Step 2** of 4

Consider the language $CUT(A)$ is defined as $CUT(A) = \{yxz \mid xyz \in A\}$.

Consider $L_1 = x,\ L_2 = y,\ L_3 = z$.

If the CFL $CUT(A)$ is closed under above operation then all conditions must be true. For showing it is not closed it will be sufficient to prove that one condition does not hold.

Comment

**Step 3** of 4

**Cyclic shift of CFL:**

Consider $L_1 \circ L_2$ exist in CFL, then it there must be Cyclic shift exist in the definition of language that is $L_2 \circ L_1$ must exist.

Consider a string $s = xyz$ in the language $CUT(A)$. It can be formed from the concatenation of the languages $X$ and $Y$ and $Z$, such that $x \in X$ and $y \in Y$ and $z \in Z$.

As per the definition of language $CUT(A)$, if $yxz \in A \text{ then } xyz \in A$. For given definition it will be closed if cyclic shift is also CFL. That is, if $xy \in A \text{ then } yx \in A$. But given condition is not true as definition of $CUT(A)$.

Comments (1)

**Step 4** of 4

The Cyclic shift of CFL which is first condition does not follow. Hence, given CFL cannot be closed under $CUT(A) = \{yxz \mid xyz \in A\}$.

Comment

## Problem

Show that every DCFG is an unambiguous CFG.

## Step-by-step solution

**Step 1** of 1

An **ambiguous grammar** is defined as "**a context free grammar** for which there subsists a string and that string may contains greater than one left-most derivation. **An unambiguous grammar** is defined as "a **context free grammar** for which all **justifiable string** has an individual leftmost derivation.

• As context free grammar (CFG's) is proper superset of deterministic context-free grammars (DCFG's). It can be derived from deterministic finite automata and it can be used to generate deterministic context free language.

• DCFG's (deterministic context-free grammars) are always shows an unambiguous behavior and an unambiguous context free grammar (CFG's) is an important super class of DCFG's.

The above statement can be proved by the following way:

• As it is known that **for every pushed down automata** $M$ **there exist an equivalent context free grammar** $G$**.**

Therefore, $M$ **Recognizes** $L\$ \Rightarrow$ $G$ **generates** $L\$$ . But, $M$ **deterministic** $\Rightarrow G$ **is an unambiguous**. Hence, **replacing** $\$$ **by** $\varepsilon$ **in** $G \Rightarrow G$ **generates** $L$.

• Thus**,** from the above explanation it can be said that **every DCFG is an unambiguous CFG.**

Comment

## Problem

Show that every DCFG generates a prefix-free language.

## Step-by-step solution

**Step 1** of 1

DCFG stands for deterministic context free grammars. It is subset of context-free grammars. DCFGs are derived from the deterministic pushdown automata (variation of pushdown automata) for generating deterministic context free languages.

Prefix-free language refers to language in which any of its members does not have any prefix. By contradiction, it can be proved that DCFG always generates prefix-free language. It is given below:

Consider there are two strings $w$ and $wz$ in $L(G)$, where $w$ and $wz$ are unequal strings, $L(G)$ is language of grammar and $G$ is DCFG. As both strings are valid, handles of them will exist. Both can be written as:

$w = xhy$ and

$$\begin{aligned} wz &= xhyz \\ &= xh\hat{y} \end{aligned}$$

'$h$' refers to handle of $w$.

Now consider $u$ and $uz$ valid strings as first reduced step of $w$ and $wz$. The process is continued till $S_1$ and $S_1z$, where $S_1$ refers to start variable. Since $S_1$ never appears at the right side, so $S_1z$ cannot be reduced. This leads to contradiction.

Comment

## Problem

Show that the class of DCFLs is not closed under the following operations:

**a.** Union

**b.** Intersection

**c.** Concatenation

**d.** Star

**e.** Reversal

## Step-by-step solution

**Step 1** of 5

a) Suppose $L_1 = \{a^i b^j c^k : i, j, k \geq 0 \, and \, i \neq j\}$ and $L_2 = \{a^i b^j c^k : i, j, k \geq 0 \, and \, i \neq k\}$ are both deterministic context free languages **(DCFL's).** However, $L_1 \cup L_2$ which is defined as $L_1 \cup L_2 = \{a^i b^j c^k : i, j, k \geq 0 \, and \, (i \neq k \, or \, i \neq j)\}$ is not comes under DCFL's which can be seen as follows.

• Suppose $L_1 \cup L_2$ is a DCFL. T hen its complement is also a DCFL as discussed in theorem. $(L_1 \cup L_2)' \cap \{a\}^* \{b\}^* \{c\}^* = \{a^i b^j c^k : i, j, k \geq 0 \, and \, (i = k = j)\}$ is a (D)CFL. This shows a **contradiction**.

• Hence, it can be said that" **Deterministic Context Free Language's (DCFL's) is not closed under union**".

Comments (1)

**Step 2** of 5

b) Non-closure under intersection follows from the non-closure under union (as discussed above) and closure under complement: $K \cup L = (K' \cap L')'$.

• Thus, if the family of DCFLs would be closed under intersection, it follows with the help of closure under complement that it would also be closed under union. This shows a **contradiction.**

• Hence, it can be said that" **Deterministic Context Free Language's (DCFL's) is not closed under intersection**".

Comment

**Step 3** of 5

c) Suppose $L_1 = \{a^i b^j c^k : i, j, k \geq 0 \, and \, i \neq j\}$ and $L_2 = \{a^i b^j c^k : i, j, k \geq 0 \, and \, i \neq k\}$ be the languages. Now, suppose $L_3 = \{d\} L_1 \cup L_2$ ,where $d$ is the symbol different from $a, b, c$. It is simple to see that $L_3 \, and \, \{d\}^*$ are **DCFL.** Now, it has intended to show that $\{d\}^* L_3$ is not a DCFL.

• Consider $\{d\}^* L_3 \cap \{d\}\{a\}^* \{b\}^* \{c\}^* = \{d\} L_1 \cup \{d\} L_2$. If $\{d\}^* L_3$ would be in DCFL then also $\{d\} L_1 \cup \{d\} L_2 = \{d\}(L_1 \cup L_2)$ is a DCFL.

• Now, it is fairly easy to see that for all words $w$ and all languages $K$ that consist $K$ are a DCFL whenever $\{w\} k$ is DCFL. Consequently, since $L_1 \cup L_2$ is not a DCFL, also $\{d\}(L_1 \cup L_2)$ is not a DCFL. This shows a **contradiction**.

• Hence, it can be said that" **Deterministic Context Free Language's (DCFL's) is not closed under concatenation**".

Comment

**Step 4** of 5

d) Suppose $L_4$ is defined as $L_4 = \{d\} \cup \{d\} L_1 \cup L_2$ where $L_1$ and $L_2$ is defined same as the above discussion.

• Consider $L_4^* \cap \{d\}^* \left(\{a\}^* \{b\}^* \{c\}^* - \{\wedge\}\right) = \{d\} L_1 \cup \{d\} L_2$. If $L_4^*$ is DCFL then also $\{d\} L_1 \cup \{d\} L_2$ should be in DCFL, but here it is not followed in this case.

• Hence, it can be said that" **Deterministic Context Free Language's (DCFL's) is not closed under star (*)"**.

Comment

**Step 5** of 5

e) Non-closure under difference follows from the closure under complement and the non-closure under intersection as discussed in part (b): $K \cap L = K - L'$, **thus if the family of DCFL's would be closed under difference.**

• The above terms follows with the help of closure under complements that would also be closed under intersection. That shows a contradiction.

• Hence, it can be said that" **Deterministic Context Free Language's (DCFL's) is not closed under reversal"**.

Comment

## Problem

Let G be the following grammar:

$$S \rightarrow T\dashv$$
$$T \rightarrow T\mathtt{a}T\mathtt{b} \mid T\mathtt{b}T\mathtt{a} \mid \varepsilon$$

**a.** Show that $L(G) = \{w\dashv \mid$ w contains equal numbers of a's and b's}. Use a proof by induction on the length of w.

**b.** Use the *DK*-test to show that G is a DCFG.

**c.** Describe a DPDA that recognizes L(G).

## Step-by-step solution

**Step 1** of 7

Consider the following grammar:

$S \rightarrow T \dashv$
$T \rightarrow TaTb \mid TbTa \mid \varepsilon$

It is required to show that $L(G) = \{w \dashv \mid w\}$

Comment

**Step 2** of 7

• Here, it can be prove by the induction that the string $w$ contains an equal number of $a$'s and $b$'s. Induction can be done in two ways either from the left side of the induction or from the right side of the induction.

• In order to prove $L(G) = \{w \dashv \mid w\}$, first $\varepsilon$ handles occurs in reduction as it is indicated by the underscores in the leftmost reduction of the string $(\ )(\ )\dashv$.

Comment

**Step 3** of 7

• It is being known that $T \rightarrow \varepsilon$ and thus $T \dashv \rightarrow \varepsilon$ and $S \rightarrow \varepsilon$.

Consider the string $w$ contains $aaabbb$. Each and every string which is being generated by the grammar *G* is in *L*.

$aaabbb \rightarrow aaSbb \rightarrow aSb \rightarrow S \rightarrow T \dashv$
$w \in T \dashv \in S \in L(G)$

Similarly in the second case, $w \dashv \in T \in S$

• Hence, it is being proved that any language $G$ contains equal number of $a$'s and $b$'s in string $w$ and $w \dashv$

Comment

**Step 4** of 7

Using the *DK*-test it is to be shown that the grammar belongs to *DCFG*.

*DK* test depend upon one simple and surprising fact. *DK* is used for accepting an input $z$ only if these two conditions are met.

• $z$ is the prefix of the valid string $v = zw$.

- $z$ should be end with a handle of $v$.

Comment

**Step 5** of 7

Consider the grammar $L(G)=\{w \dashv | w\}$ which contains an equal number of $a$'s and $b$'s.

- So, if the grammar contains an equal number of $a$'s and $b$'s then the grammar $a^n b^n$ becomes $a^n b^n$. In case of *DCFG* all the handles are being forced. So, if $zw$ is a one of the valid string with the prefix $z$ that will end for handling $zw$'.

- In case of *DCFG* all the handles are being forced. So, if $zw$ is a one of the valid string with the prefix $z$ that will end for handling $zw$'. For all this property *DK* must be capable of handling all the accept states. But, the condition is that outgoing path should not be there.

Comment

**Step 6** of 7

- Here, the construction of *DK* test is done for concluding that *G* is deterministic if the property of accept state is satisfied. For the construction of the *DFA*, first *NFA* need to be constructed. After the construction it is found that no outgoing path is there from the start state.

- Here, $aaabbb$ is a valid string and it handles $zw$ so after the reduction of the rule it becomes: $T \to z.w$. Grammar must contain the following rules:

$$
\begin{aligned}
&S_1 \to z_1 S_2 w_1 \\
&S2 \to z_2 S_3 w_2 \\
&\quad\vdots \\
&S_l \to z_l T w_l \\
&T \to zw
\end{aligned}
$$

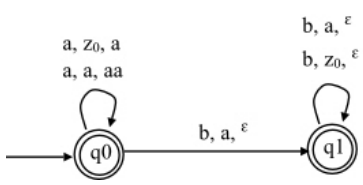
*K* contains the path from the start state to the end state. First the input $z = xu$ is being read after that transition is done and shifting of the string is done. After the shifting it is being found that there is no outgoing path from the start state.

Hence with the help of *DK* test it is proved that grammar belong to *DCFG*.

Comment

**Step 7** of 7

The DPDA that recognizes the L(G) is as follows:




Comment

## Problem

Let $G_1$ be the following grammar that we introduced in Example 2.45. Use the *DK*-test to show that $G_1$ is not a DCFG.

$$R \rightarrow S \mid T$$
$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \mathtt{ab}$$
$$T \rightarrow \mathtt{a}T\mathtt{bb} \mid \mathtt{abb}$$

## Step-by-step solution

**Step 1** of 3

$DK$ **-test is a procedure which is exactly used to determine whether a CFG is deterministic or not**. $DK$ -test comes into picture because the definition of DCFG's does not give any accurate way to determine whether a CFG is deterministic. The procedure of DK-test can be explained as follow.

Starting with a CFG $G$ , construct the associated DFA $DK$ .Now, determine whether G is deterministic by testing $DK's$ accept states. The $DK$ -test stipulates that every accept state contains:

1. **Exactly one completed rule, and**

2. **No dotted rule in which terminal symbol immediately follows the dot (that is, no dotted rule of the form** $B \rightarrow u.av \; for \; a \in \sum$ ).

Comment

**Step 2** of 3

Now**, consider the grammar** $G_1$ which is given below:

$R \rightarrow S \mid T$
$S \rightarrow aSb \mid ab$
$T \rightarrow aTbb \mid abb$

• **The language of grammar** is $B \cup C$ where $B = \{a^m b^m \mid m \geq 1\} \; and \; C = \{a^m b^{2m} \mid m \geq 1\}$. In the leftmost reduction of string $aaabbb \in L(G_1)$. It is underlined the handle at each step:

$aa\underline{ab}bb \rightarrow a\underline{aSb}b \rightarrow \underline{aSb} \rightarrow \underline{S \rightarrow R}$

• **Equivalently**, this is leftmost reduction of the string $aaabbbbbb$ :

$aa\underline{abb}bbbb \rightarrow a\underline{aTbb}bb \rightarrow \underline{aTbb} \rightarrow \underline{T} \rightarrow R$.

• In the both of the cases which is given above, the leftmost reduction as shown happens to be **the only reduction possible**. In other grammars, where many reduction may occur.

• Here, it is notice that the handles of $aaabbb$ and $aaabbbbbb$ **are unequal**, even though the initial part of these strings agree.

Comment

**Step 3** of 3

**Therefore**, from the above discussion **it violates the condition of** $DK$ **-test (as** $DK$ **-test say that every accept state must consist of exactly one completed rule**), because **so many completed rules present here**. **Thus,** it can be said that "**The grammar** $G_1$ **is not a DCFG".**

Comment

## Problem

Let A = L($G_1$) where $G_1$ is defined in Problem 2.55. Show that A is not a DCFL.

(Hint: Assume that A is a DCFL and consider its DPDA P. Modify P so that its input alphabet is {a, b, c}. When it first enters an accept state, it pretends that c's are b's in the input from that point on. What language would the modified P accept?)

Problem 2.55

Let $G_1$ be the following grammar that we introduced in Example 2.45. Use the *DK*-test to show that $G_1$ is not a DCFG.

$$\begin{aligned} R &\rightarrow S \mid T \\ S &\rightarrow \mathtt{a}S\mathtt{b} \mid \mathtt{ab} \\ T &\rightarrow \mathtt{a}T\mathtt{bb} \mid \mathtt{abb} \end{aligned}$$

## Step-by-step solution

**Step 1** of 3

**On the contrary suppose *A* is a deterministic context free language**. Consider *p* as the pumping length of *A*, such that *p* length string of *A* will satisfy the pumping lemma. Consider '*m*' as string of *A* with $m = 0^{2p}0^p1^p0^{2p}$. In order to satisfy the assumption, there are following ways so that '*w*' can be written as $uvxyz$, where $|vy| \geq 1$ and $|vxy| \leq p$, $uv^ixy^iz$ string lies in *A* for any value of *i*.

Comment

**Step 2** of 3

**On the basis of above condition there are only 3 cases which are as follow:**

**Case 1:**

In string $uv^ixy^iz$, $vy$ only have 0s which are getting from the last $0^{2p}$ of *m*. Assume that '*i* 'is any number which satisfy the condition $7p > |vy| \times (i+1) \geq 6p$. Then, may be the length of $uv^ixy^iz$ is not become the multiple of 3, or may be the string is in the form of $wtw'$ specified that $|w| = |t| = |w'|$ having all zero's in $w'$ and not all zero's in $w$ $(\text{Means}, w^R \neq w')$.

**Case 2:**

In string $uv^ixy^iz$, $vy$ does not have 0s in last $0^{2p}$ of *m*. Then, may be the length of $uv^2xy^2z$ is not become the multiple of 3, or may be the string is in the form of $wtw'$ specified that $|w| = |t| = |w'|$ having all zero's in $w$ and not all zero's in $w'$ $(\text{Means}, w^R \neq w')$.

Comment

**Step 3** of 3

**Case 3:**

In string $uv^ixy^iz$, $vy$ have some 0s which are getting from the last $0^{2p}$ of *m*. In this case $|vxy| \leq p$, therefore $vxy$ must be a substring $1^p0^p$. Then, may be the length of $uv^2xy^2z$ is not become the multiple of 3, or may be the string is in the form of $wtw'$ specified that $|w| = |t| = |w'|$ having all zero's in $w$ and not all zero's in $w'$ $(\text{Means}, w^R \neq w')$.

By all these case it is observes that *m* cannot satisfy the conditions of the assumption. Therefore, a contradiction occurs. Hence, it can be state that *A* is not a deterministic context free language.

Comment

## Problem

Let $B = \{\mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$. Prove that B is not a DCFL.

## Step-by-step solution

**Step 1** of 1

The following facts will be used to proof " $B = \left\{\mathbf{a^i b^j c^k} \mid \mathbf{i, j, k \geq 0 \text{ and } i = j \text{ or } i = k}\right\}$ is not a DCFL (Deterministic Context Free Language).

- $B$ is context free.
- If $B$ is deterministic context free then $\overline{B} = \{a, b, c\}^* - B$ is deterministic context free.
- $B_1 = \left\{a^i b^j c^k \mid i \neq j \text{ and } j \neq k\right\}$ is not context free .

Here, it is recorded that $\overline{B} \cap a^* b^* c^* = B_1$. **Suppose that $\overline{B}$ was a DCFL (Deterministic Context Free Language) implying that $\overline{B}$ must be a deterministic context free language.** Which in turn implies that $\overline{B}$ is a context free language (CFL). As the **intersection of a CFL and Regular Language is a CFL.** However, $B_1$ is not a context free language (CFL) that shows a contradiction. **Therefore, $B$ is not Deterministic Context Free Language (DCFL**).

Comment

## Problem

Let $C = \{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$. Prove that C is not a DCFL. (Hint: Suppose that when some DPDA P is started in state q with symbol x on the top of its stack, P never pops its stack below x, no matter what input string P reads from that point on. In that case, the contents of P's stack at that point cannot affect its subsequent behavior, so P's subsequent behavior can depend only on q and x.)

## Step-by-step solution

**Step 1** of 2

A deterministic context-free language (DCFL) is defined as a language for which there is a DPDA that accepts the same language. Consider the language which is given below:

**Language** $L_0 = \left\{ww^R \mid w \in \{0,1\}^*\right\}$

**The language $L_0$ which is given above is not deterministic context free language because any pushed down automata (PDA), the use of non-determinism is essential to "guess" the midpoint.**

Comment

**Step 2** of 2

**Consider the table which is given below:**

| Current State | Input | Stack top | String pushed | Current State | Description |
|---|---|---|---|---|---|
| q0 | 0 | z0 | 0z0 | q1 | |
| q0 | 1 | z0 | 1z0 | q1 | 0. Have to push on this one |
| q1 | 0 | 0 | 00 | q1 | …or this one |
| q1 | 0 | 1 | 01 | q1 | 1a: Suppose not at midpoint have to push on this one. |
| q1 | 0 | 0 | $\in$ | q1 | 1b: Suppose at the midpoint |
| q1 | 1 | 1 | 11 | q1 | 2a: Suppose not at midpoint have to push on this one. |
| q1 | 1 | 0 | 10 | q1 | 2b:Assume at the midpoint |
| q1 | 1 | 1 | $\in$ | q1 | 3. Matched around midpoint |
| q1 | $\in$ | z0 | z0 | q2 | |

The above table explained the $\partial$ function of **pushed down automata (PDA), which is designed to recognize $L_0$.**

• The given PDA is described by the structure:

$$P_{L_0} = \left(\{q_0\},\{0,1\},\{0,1,z_0\},\partial,q_0,z_0,\{q_0,q_2\}\right)$$

The given PDA starts by stacking 0 or 1, depending on which one comes first. The comment 1a and 1b explain the non-deterministic selection of assuming not being at a midpoint, and being a midpoint, respectively. For 2a and 2b similar logic is followed as well.

• Therefore, from the above explanation it can be said that **the language $L_0$, which is given by $\mathbf{L_0 = \left\{ww^R \mid w \in \{0,1\}^*\right\}}$, is not a Deterministic Context Free Language (DCFL).**

Comment

## Problem

If we disallow "-rules in CFGs, we can simplify the *DK*-test. In the simplified test, we only need to check that each of *DK*'s accept states has a single rule. Prove that a CFG without "-rules passes the simplified *DK*-test iff it is a DCFG.

## Step-by-step solution

**Step 1** of 1

**Given:**

Consider a context free grammar *C* which does not have $\varepsilon$ rule. Apply DK-test on *C* to check whether it is a deterministic CFG or not.

**DK Test:**

This test is used to check whether CFG is deterministic CFG or not. Consider a CFG C, create DK an associated DFA. Check whether CFG is deterministic by checking the accept state of DK. Every accept states must have:

• Only 1 completed rule.

• There should be no dotted rule that is mean dot should not be immediately after the terminal.

**Proof:**

Assume a contrary that C is not a deterministic CFG to show the failure of DK-test. Consider a valid string $ahb$ which has *h* as unforced handle. It may possible that some other valid string $ahb'$ has another handle $\hat{h} \neq h$. Therefore $ahb'$ can be rewrite as $\hat{a}\hat{h}\hat{b}$ where $b'$ is a terminal.

• When $ah = \hat{a}\hat{h}$, this lead to change the reduce rule because $h$ and $\hat{h}$ both are two different handle. Hence, $ah$ take the DK to state which has two completed rule which does not satisfy the DK-test.

• When $ah \neq \hat{a}\hat{h}$, the one extends the other. Assume that proper prefix of $\hat{a}\hat{h}$ is $ah$. There are same arguments with the interchanged string and in place of $b'$ use $b$.

Assume that *w* is an accepting state in which DK enters on $ah$ input. The state *w* must be an accepting states because $h$ is the handle of $ahb$. As *w* is the accepting state, so the transition arrow must terminate at w also $\hat{a}\hat{h}$ take DK to accepting state by *w*.

The transition must label with $b'$ because $b' \in \Sigma^+$ also *C* does not have null rule. Hence *w* have dotted rule that is, dot exist immediately after terminal symbol which does not satisfy the DK-test.

**Conclusion:**

The DK test fails; therefore contradiction occurs that C is not a deterministic CFG. Hence CFG is a DCFG.

Comment

## Problem

This exercise concerns TM $M_2$, whose description and state diagram appear in Example 3.7. In each of the parts, give the sequence of configurations that $M_2$ enters when started on the indicated input string.

**a.** 0.

A**b.** 00.

**c.** 000.

**d.** 000000.

**EXAMPLE 3.7**

Here we describe a Turing machine (TM) $M_2$ that decides $A = \{0^{2^n} \mid n \geq 0\}$, the language consisting of all strings of 0s whose length is a power of 2.

$M_2$ = "On input string $w$:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, *accept*.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, *reject*.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."

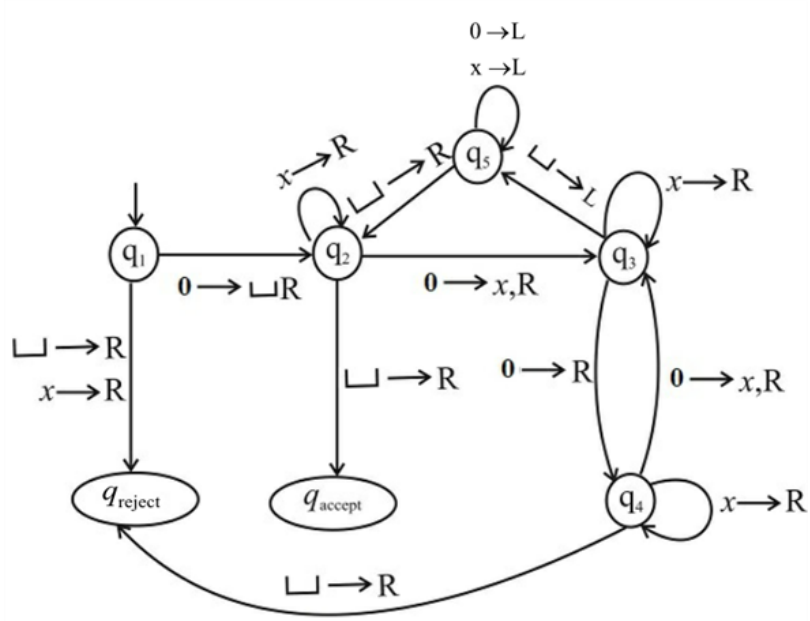
## Step-by-step solution

**Step 1** of 7

Consider the language $A = \{0^{2^n} \mid n \geq 0\}$, consisting of all strings of 0s whose length is a power of 2.

Turing machine $M_2$ decides a language A.

The state diagram for $M_2$ is as follows.



Comment

**Step 2** of 7

In this state diagram the label $0 \to x, R$, appears on the transition from $q_4$ to $q_3$. This label signifies that, the state $q_4$ with head reading 0, the machine goes to state $q_3$, writes *x*, and moves the head to the right. In the similar manner, other transitions also occur.

Comment

**Step 3** of 7

a. 0

Run the machine $M_2$ on the input 0. The starting configuration is $q_1 0$.The sequence of configurations that the machine enters when started on the input string is as follows:

$$\begin{array}{ll} q_1 0 & (\text{At } q_1, 0 \to \sqcup, \text{R goes to the } q_2) \\ \sqcup q_2 \sqcup & (\text{At } q_2, \sqcup \to \text{R goes to the accept state}) \\ \sqcup \sqcup q_{accept} & \end{array}$$

The state $q_1$ on 0, the machine goes to state $q_2$, writes $\sqcup$ and moves the head to the right. The state $q_2$ on $\sqcup$, the machine goes to state $q_{accept}$, then halts.

**As $M_2$ enters $q_{accept}$ state the input 0 is accepted.**

Comment

**Step 4** of 7

b. 00

Run the machine $M_2$ on the input 00. The starting configuration is $q_1 00$.The sequence of configurations that the machine enters when started on the input string is as follows:

$$\begin{array}{ll} q_1 00 & (\text{At } q_1, 0 \to \sqcup, \text{R goes to the } q_2) \\ \sqcup q_2 0 & (\text{At } q_2, 0 \to \text{ x, R goes to the } q_3) \\ \sqcup x q_3 \sqcup & (\text{At } q_3, \sqcup \to \text{L goes to the } q_5) \\ \sqcup q_5 x \sqcup & (\text{At } q_5, x \to \text{L goes to the } q_5) \\ q_5 \sqcup x \sqcup & (\text{At } q_5, \sqcup \to \text{R goes to the } q_2) \\ \sqcup q_2 x \sqcup & (\text{At } q_2, x \to \text{R goes to the accept state}) \\ \sqcup x q_2 \sqcup & (\text{At } q_2, \sqcup \to \text{R goes to the accept state}) \\ \sqcup x \sqcup q_{accept} & \end{array}$$

The state $q_1$ on 0, the machine goes to state $q_2$, writes $\sqcup$ and moves the head to the right. The state $q_2$ on 0, the machine goes to state $q_3$, writes x and moves the head to the right. The state $q_3$ on $\sqcup$, the machine goes to state $q_5$, moves the head to the left. The state $q_5$ on x, the machine goes to state $q_5$, moves the head to the left. The state $q_5$ on $\sqcup$, the machine goes to state $q_2$, moves the head to the right. The state $q_2$ on x, the machine goes to state $q_2$ itself, moves the head to the right. The state $q_2$ on $\sqcup$, the machine goes to state $q_{accept}$, then halts.

**Finally, $M_2$ enters $q_{accept}$ state. Thus, the input 00 is accepted.**

Comments (2)

**Step 5** of 7

c. 000

Run the machine $M_2$ on the input 000. The starting configuration is $q_1 000$.The sequence of configurations that the machine enters when started on the input string is as follows:

Comment

**Step 6** of 7

$$
\begin{array}{ll}
q_1 000 & (\text{At } q_1, 0 \to \sqcup, R \text{ goes to the } q_2) \\
\sqcup q_2 00 & (\text{At } q_2, 0 \to x, R \text{ goes to the } q_3) \\
\sqcup x q_3 0 & (\text{At } q_3, 0 \to R \text{ goes to the } q_4) \\
\sqcup x 0 q_4 \sqcup & (\text{At } q_4, \sqcup \to R \text{ goes to the } q_{reject}) \\
\sqcup x 0 \sqcup q_{reject} &
\end{array}
$$

The state $q_1$ on 0, the machine goes to state $q_2$, writes $\sqcup$ and moves the head to the right. The state $q_2$ on 0, the machine goes to state $q_3$, writes x and moves the head to the right. The state $q_3$ on 0, the machine goes to state $q_4$, moves the head to the right. The state $q_4$ on $\sqcup$, the machine goes to state $q_{reject}$, moves the head to the right.

**Finally, $M_2$ enters $q_{reject}$ state. Thus, input 000 is rejected.**

Comment

**Step 7** of 7

d. 000000

Run the machine $M_2$ on the input 000000. The starting configuration is $q_1 000000$.The sequence of configurations that the machine enters when started on the input string is as follows:

$q_1 000000$ (At $q_1$, $0 \to \sqcup$, R goes to the $q_2$)

$\sqcup q_2 00000$ (At $q_2$, $0 \to$ x, R goes to the $q_3$)

$\sqcup x q_3 0000$ (At $q_3$, $0 \to$ R goes to the $q_4$)

$\sqcup x 0 q_4 000$ (At $q_4$, $0 \to$ x, R goes to the $q_3$)

$\sqcup x 0 x q_3 00$ (At $q_3$, $0 \to$ R goes to the $q_4$)

$\sqcup x 0 x 0 q_4 0$ (At $q_4$, $0 \to$ x, R goes to the $q_3$)

$\sqcup x 0 x 0 x q_3 \sqcup$ (At $q_3$, $\sqcup \to$ L goes to the $q_5$)

$\sqcup x 0 x 0 q_5 x \sqcup$ (At $q_5$, x $\to$ L goes to the $q_5$)

$\sqcup x 0 x q_5 0 x \sqcup$ (At $q_5$, 0 $\to$ L goes to the $q_5$)

$\sqcup x 0 q_5 x 0 x \sqcup$ (At $q_5$, x $\to$ L goes to the $q_5$)

$\sqcup x q_5 0 x 0 x \sqcup$ (At $q_5$, 0 $\to$ L goes to the $q_5$)

$\sqcup q_5 x 0 x 0 x \sqcup$ (At $q_5$, x $\to$ L goes to the $q_5$)

$q_5 \sqcup x 0 x 0 x \sqcup$ (At $q_5$, $\sqcup \to$ R goes to the $q_2$)

$\sqcup q_2 x 0 x 0 x \sqcup$ (At $q_2$, x $\to$ R goes to the $q_2$)

$\sqcup x q_2 0 x 0 x \sqcup$ (At $q_2$, 0 $\to$ x, R goes to the $q_3$)

$\sqcup x x q_3 x 0 x \sqcup$ (At $q_3$, x $\to$ R goes to the $q_3$)

$\sqcup x x x q_3 0 x \sqcup$ (At $q_3$, 0 $\to$ R goes to the $q_4$)

$\sqcup x x x 0 q_4 x \sqcup$ (At $q_4$, x $\to$ R goes to the $q_4$)

$\sqcup x x x 0 x q_4 \sqcup$ (At $q_4$, $\sqcup \to$ R goes to the $q_{reject}$)

$\sqcup x x x 0 x \sqcup q_{reject}$

**Finally, $M_2$ enters $q_{reject}$ state. Hence, the input 000000 is rejected.**

Comment

## Problem

This exercise concerns TM $M_1$, whose description and state diagram appear in Example 3.9. In each of the parts, give the sequence of configurations that $M_1$ enters when started on the indicated input string.

**Aa.** 11.

**b.** 1#1.

**c.** 1##1.

**d.** 10#11.

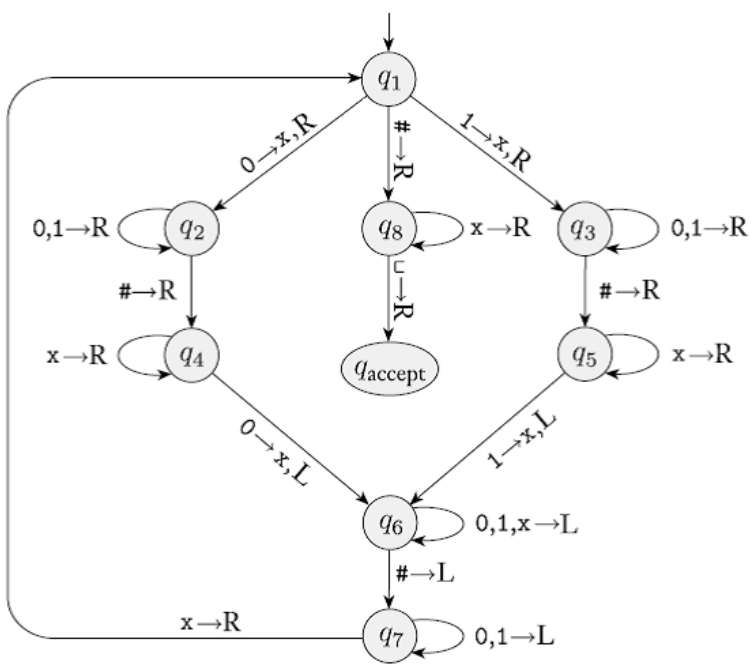
**e.** 10#10.

Example 3.9

**EXAMPLE 3.9**

The following is a formal description of $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$, the Turing machine that we informally described (page 167) for deciding the language $B = \{w\#w \mid w \in \{0,1\}^*\}$.

- $Q = \{q_1, \ldots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0,1,\#\}$, and $\Gamma = \{0,1,\#,x,\sqcup\}$.
- We describe $\delta$ with a state diagram (see the following figure).
- The start, accept, and reject states are $q_1$, $q_{\text{accept}}$, and $q_{\text{reject}}$, respectively.




## Step-by-step solution

**Step 1** of 6

Consider the language $B = \{w\#w \mid w \in \{0,1\}^*\}$. The Turing machine $M_1$ that decides the language B.

In the state diagram, reject state has not shown for simplicity. If the state does not have an outgoing transition for any symbol, then it moves to the reject state $q_{\text{reject}}$.

In the state diagram, the label $0 \to x, R$ appears on the transition from $q_1$ to $q_2$. This label signifies that, the state $q_1$ with head reading 0, the machine goes to state $q_2$, writes *x*, and moves the head to the right.

Comment

**Step 2** of 6

a.

Consider the input string 11. The sequence of configurations that $M_1$ enters are as follows:

$$\begin{aligned}
&\to q_1 11 \\
&\to \mathrm{x} q_3 1 \\
&\to \mathrm{x} 1 q_3 \sqcup \quad [\because\ q_3 \text{ is not reading} \sqcup, \text{ so it enters to reject state}] \\
&\to \mathrm{x} 1 \sqcup q_{\text{reject}}
\end{aligned}$$

Finally, $M_1$ enters the $q_{\text{reject}}$ state. Hence input 11 is rejected.

Comment

**Step 3** of 6

b.

Consider the input string 1#1. The sequence of configurations that $M_1$ enters are as follows:

$$\begin{aligned}
&\to q_1 1\#1 \\
&\to \mathrm{x} q_3 \#1 \\
&\to \mathrm{x}\#q_5 1 \\
&\to \mathrm{x} q_6 \#\mathrm{x} \\
&\to q_7 \mathrm{x}\#\mathrm{x} \\
&\to \mathrm{x} q_1 \#\mathrm{x} \\
&\to \mathrm{x}\#q_8 \mathrm{x} \\
&\to \mathrm{x}\#\mathrm{x} q_8 \sqcup \\
&\to \mathrm{x}\#\mathrm{x} \sqcup q_{\text{accept}}
\end{aligned}$$

Finally, $M_1$ enters $q_{\text{accept}}$ state. Thus, the input 1#1 is accepted.

Comment

**Step 4** of 6

c.

Consider the input string 1##1. The sequence of configurations that $M_1$ enters are as follows:

$$\begin{aligned}
&\to q_1 1\#\#1 \\
&\to \mathrm{x} q_3 \#\#1 \\
&\to \mathrm{x}\#q_5 \#1 \quad [\because\ \mathrm{q}_5 \text{ is not reading } \#, \text{ so it enters to reject state}] \\
&\to \mathrm{x}\#\#q_{\text{reject}} 1
\end{aligned}$$

Finally, $M_1$ enters $q_{reject}$ state. Thus, the input 1##1 is rejected.

Comment

**Step 5** of 6

d.

Consider the input string 10#11. The sequence of configurations that $M_1$ enters are as follows:

$$\begin{aligned}
&\to q_1 10\#11\\
&\to \mathrm{x}q_3 0\#11\\
&\to \mathrm{x}0q_3\#11\\
&\to \mathrm{x}0\#q_5 11\\
&\to \mathrm{x}0q_6\#\mathrm{x}1\\
&\to \mathrm{x}q_7 0\#\mathrm{x}1\\
&\to q_7\mathrm{x}0\#\mathrm{x}1\\
&\to \mathrm{x}q_1 0\#\mathrm{x}1\\
&\to \mathrm{xx}q_2\#\mathrm{x}1\\
&\to \mathrm{xx}\#q_4\mathrm{x}1\\
&\to \mathrm{xx}\#\mathrm{x}q_4 1\ [\because q_4 \text{is not reading 1, so it enters to reject state}]\\
&\to \mathrm{xx}\#\mathrm{x}1q_{\text{reject}}
\end{aligned}$$

Finally, $M_1$ enters $q_{reject}$ state. Thus, the input 10#11 is rejected.

Comment

**Step 6** of 6

e.

Consider the input string 10#10. The sequence of configurations that $M_1$ enters are as follows:

$\rightarrow q_1 10\#10$

$\rightarrow \mathrm{x}q_3 0\#10$

$\rightarrow \mathrm{x}0q_3\#10$

$\rightarrow \mathrm{x}0\#q_5 10$

$\rightarrow \mathrm{x}0q_6\#\mathrm{x}0$

$\rightarrow \mathrm{x}q_7 0\#\mathrm{x}0$

$\rightarrow q_7\mathrm{x}0\#\mathrm{x}0$

$\rightarrow \mathrm{x}q_1 0\#\mathrm{x}0$

$\rightarrow \mathrm{xx}q_2\#\mathrm{x}0$

$\rightarrow \mathrm{xx}\#q_4\mathrm{x}0$

$\rightarrow \mathrm{xx}\#\mathrm{x}q_4 0$

$\rightarrow \mathrm{xx}\#q_6\mathrm{xx}$

$\rightarrow \mathrm{xx}q_6\#\mathrm{xx}$

$\rightarrow \mathrm{x}q_7\mathrm{x}\#\mathrm{xx}$

$\rightarrow \mathrm{xx}q_1\#\mathrm{xx}$

$\rightarrow \mathrm{xx}\#q_8\mathrm{xx}$

$\rightarrow \mathrm{xx}\#\mathrm{x}q_8\mathrm{x}$

$\rightarrow \mathrm{xx}\#\mathrm{xx}q_8\sqcup$

$\rightarrow \mathrm{xx}\#\mathrm{xx}\sqcup q_{\mathrm{accept}}$

Finally, $M_1$ enters $q_{\text{accept}}$ state. Thus, the input 10#10 is accepted.

Comment

## Problem

Modify the proof of Theorem 3.16 to obtain Corollary 3.19, showing that a language is decidable iff some nondeterministic Turing machine decides it. (You may assume the following theorem about trees. If every node in a tree has finitely many children and every branch of the tree has finitely many nodes, the tree itself has finitely many nodes.)

THEOREM **3.16**

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

COROLLARY **3.19**

A language is decidable if and only if some nondeterministic Turing machine decides it.

## Step-by-step solution

**Step 1** of 4

As per the theorem 3.16, Every DTM (Deterministic Turing machine) has an equivalent NDTM (Non-deterministic Turing machine). The corollary 3.19, states that a language is decidable iff some NDTM (Non-deterministic Turing machine) decides it.

In order to modify the proof of theorem 3.16 to obtain corollary 3.19, prove the following:

(i) If a Language *L* is decidable then the language *L* can be decided by NDTM (non-deterministic Turing machine).

Let *L* be a language. The Language *L* is decidable if it is decided by a deterministic Turing machine. Any DTM (Deterministic Turing machine) is automatically a NDTM (Non-deterministic Turing machine).

Thus, language L is decidable if it is decided by NDTM (Non-deterministic Turing machine) TM also.

Comment

**Step 2** of 4

(ii) If some NDTM (Non-deterministic Turing machine) decides a language then it is decidable.

Let *L* be an any language and it is decided by NDTM (Non-deterministic Turing machine).

Construct a deterministic TM *D* that decides *L* and prove that *D* simulates *N*.

Construction of *D* and Simulation of *D* with *N*:

The simulating deterministic TM '*D*' has 3 tapes

1. Input tape

2. Simulating tape

3. Address tape

By the theorem

"Every multi tape TM has an equivalent single tape TM"

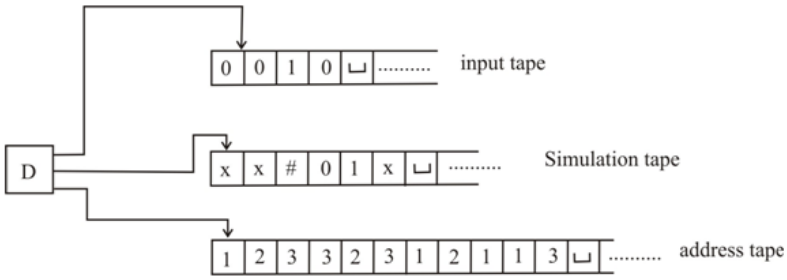
*D*'s arrangement is equivalent to having a single tape TM.

The machine *D* uses its three tapes in the following way:

• Tape 1 always contains the input string and is never altered.

• Tape 2 maintains a copy of *N*'s tape on some branch of its non-deterministic computation

• Tape 3 keeps track of *D*'s location in N's non-deterministic computation tree.



Comment

**Step 3** of 4

The description of $D$ in detail as follows:

**Stage 1:** Initially tape 1 contains the input $w$, and tapes 2 and 3 are empty.

**Stage 2:** Copy tape 1 to tape 2.

**Stage 3:** Use tape 2 to simulate $N$ with input $w$ on one branch of its non-deterministic computation. Before each step of $N$ check the next symbol on tape 3 to determine which choice to make among those allowed by $N$'s transition function. If no more symbols left on tape 3 or if this non-deterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, then accept the input.

**Stage 4:** Replace the string on tape 3 with the lexicographically next string. Simulate the next branch of $N$'s computation by going to stage 2.

**Stage 5:** Reject if all branches of non-determinism of $N$ are exhausted.

Comment

**Step 4** of 4

Now it can be said that $D$ is a decider of $L$.

• If $N$ accepts its input, then $D$ will find an accepting branch and accept.

• If $N$ rejects its input, then all its branches halt and reject.

Thus, the language $L$ is decidable.

From (i) and (ii),

**The language $L$ is decidable if and only if some non-deterministic Turing machine decides it.**

Comment

## Problem

Give a formal definition of an enumerator. Consider it to be a type of two-tape Turing machine that uses its second tape as the printer. Include a definition of the enumerated language.

## Step-by-step solution

**Step 1** of 3

An enumerator is Turing machine(TM) variant with as attached printer. The enumerator uses the printer as output device to print strings. Every time Turing machine wants to add a string to the list, it sends the string to the printer.

**Formal definition of an enumerator:**

An Enumerator is a 7-tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, q_{print}, q_{halt} \rangle$

Where $Q, \Sigma, \Gamma$ are all finite sets and

(i) $Q$ is the set of states

(ii) $\Gamma$ is the work tape alphabet,

(iii) $\Sigma$ is the output/print tape alphabet

(iv) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\} \times \Sigma_{\in}$ is the transition function,

(v) $q_0 \in Q$ is the start state

(vi) $q_{print} \in Q$ is the print state, and

(vii) $q_{halt} \in Q$ is the halt state, where $q_{print} \neq q_{halt}$

Comment

**Step 2** of 3

**Computation of an enumerator:**

The computation of an enumerator E is defined as an ordinary TM, except for following three points.

1. It has two tapes,

• First tape is work tape and

• Second tape is print tape

Both are initially blank.

2. At each step, the machine may write a symbol from $\Sigma$ on the output tape, or nothing, as determined by $\delta$.

For example,

If $\delta(q_{prev}, a) = (q_{next}, b, L, c)$, it means that in state $q_{prev}$, reading a, enumerator E enters state $q_{next}$, writes *b* on the work tape moves the work tape head left and writes *c* on the print tape, and moves the print tape head to the right if $c \neq \in$.

3. Whenever state $q_{print}$ is entered, the output tape is reset to blank and the head returns to the left hand end and the output is printed.

When $q_{Halt}$ is entered the machine will halt.

Comment

**Step 3** of 3

**Enumerated language:**

The language enumerated by Enumerator is the collection of all the strings that Enumerator eventually prints out.

Enumerator may generate the strings of the language in any order, possibly with repetitions.

*L(E):* be the language enumerated by enumerator E

Thus *L(E)* is the language that contains the strings which are printed out.

$$\boxed{L(E) = \{w \in \Sigma^* \mid w \text{ appears on the work tape if } q_{\text{print}} \text{ is entered}\}}$$

Comment

## Problem

Examine the formal definition of a Turing machine to answer the following questions, and explain your reasoning.

**a.** Can a Turing machine ever write the blank symbol $\sqcup$ on its tape?

**b.** Can the tape alphabet $\Gamma$ be the same as the input alphabet $\sum$?

**c.** Can a Turing machine's head *ever* be in the same location in two successive steps?

**d.** Can a Turing machine contain just a single state?

## Step-by-step solution

**Step 1** of 4

(a) **Yes.** A Turing machine can write the blank symbol $\square$ on its tape.

According to the definition, a Turing machine is a 7 – tuple $\left(Q,\Sigma,\Gamma,\delta,q_0,q_{accept},q_{reject}\right)$

Where $Q,\Sigma,\Gamma$ are all finite sets.

$\Sigma$ is the input alphabet not containing the blank symbol $\square$.

But $\Gamma$ is the tape alphabet, where $\square \in \Gamma$ and $\Sigma \subseteq \Gamma$.

A Turing machine can write any characters in $\Gamma$ on its tape.

Thus Turing machine can write blank symbol on its tape.

Comment

**Step 2** of 4

(b) **No.** The tape alphabet $\Gamma$ never same as input alphabet $\Sigma$

Because tape alphabet $\Gamma$ contains blank symbol $\square$ where as input alphabet does not contain blank symbol.

Always $\Sigma$ is subset of $\Gamma$, but never same as $\Sigma$.

Comment

**Step 3** of 4

(c) **Yes.** Turing machines head be in the same location in two successive steps.

When the Turing machines head is at the left _end of the tape, if Turing machine again try to move left side then Turing machine's head is in the same location as previous can move.

Comment

**Step 4** of 4

(d) **No.**

According to definition, A Turing machine is a 7- tuple ( $\left(Q,\Sigma,\Gamma,\delta,q_0,q_{accept},q_{reject}\right)$

Where $Q$ is the set of states.

$q_{accept}$ is the accept state and is belongs to $Q$

$q_{reject}$ is the reject state and is belongs to $Q$.

But $q_{accept} \neq q_{reject}$

Thus there must be two distinct states $q_{accept}$ and $q_{reject}$

So, a Turing machine contains at least two states.

Comment

## Problem

In Theorem 3.21, we showed that a language is Turing-recognizable iff some enumerator enumerates it. Why didn't we use the following simpler algorithm for the forward direction of the proof? As before, $s_1, s_2, \ldots$ is a list of all strings in $\Sigma^*$.

E = "Ignore the input.

**1.** Repeat the following for i = 1, 2, 3, . . . .

**2.** RunM on $s_i$.

**3.** If it accepts, print out $s_i$."

THEOREM **3.21**

A language is Turing-recognizable if and only if some enumerator enumerates it.

## Step-by-step solution

**Step 1** of 1

**Theorem: -** A language is Turing – recognizable if and only if some enumerator enumerates it.

The given simpler algorithm for the forward direction of the proof of this theorem is

Say that $s_1, s_2 \cdots$ is a list of all strings in $\Sigma^*$

If Turing Machine M recognizes a language L, then we can construct following enumerator E for L.

The Enumerator E works as follows:

$E = \text{"Ignore the input}$

1. Repeat the following for $i = 1, 2, 3, \ldots$

2. Run M on $S_i$;

3. If it accepts, print out $S_i$"

**Defects in this proof:**

In stage 2 of this algorithm (Run M on $S_i$)

If M loops on a certain input $S_i$ runs forever, E could not check any input after $S_i$.

If it occurs, then E might fail to enumerate its language L as required.

Thus this procedure does not give the effect of running M in parallel on all possible input strings.

So, this proof is not suited for forward direction of above theorem.

Comment

## Problem

Explain why the following is not a description of a legitimate Turing machine.

$M_{bad}$ = "On input $\langle p \rangle$, a polynomial over variables $x_1, \ldots, x_k$:

**1.** Try all possible settings of $x_1, \ldots, x_k$ to integer values.

**2.** Evaluate p on all of these settings.

**3.** If any of these settings evaluates to 0, *accept* ; otherwise, *reject* ."

## Step-by-step solution

**Step 1** of 2

Consider the steps for the legitimate turing machine:

$M_{bad}$ =" The input is a polynomial P over variables $x_1, \ldots, x_k$

Step 1: Try all possible values of $x_1, \ldots, x_k$ to integer values.

Step 2: Evaluate P on these values

Step 3: If any of these settings evaluates to 0, accept; otherwise, reject"

• From the step 1 to store all the values turing machine require infinite memory location. So, this is not possible to accept the machine.

• For the Step 2, infinite processing time to require all the values for evaluation. So, this step also not possible to accept the machine.

• If the above two steps are not possible, then the step 3 is rejected.

Comment

**Step 2** of 2

So, the Turing machine $M_{bad}$ could require **infinite time and infinite steps** to try all of them.

But the above settings are required that every stage in the Turing machine description be completed in a finite number of steps.

Thus, this description is not suitable for the legitimate Turing machine.

Comment

## Problem

Give implementation-level descriptions of Turing machines that decide the following languages over the alphabet {0,1}.

A**a.** {wl w contains an equal number of 0s and 1s}

**b.** {wl w contains twice as many 0s as 1s}

**c.** {wl w does not contain twice as many 0s as 1s}

## Step-by-step solution

**Step 1** of 3

**Turing decidable: I**f some Turing machine decides the language then it is said to be Turing decidable.

a.

Given language $L_1$ is { $w \mid w$ contains an equal number of 0's and 1's} over the alphabet $\{0,1\}$.

Let $M_1$ be the Turing machine that decides the language L. The implementation level description of $M_1$ is as follows:

$M_1 =$" on input string $w$:

**Step 1:**

Mark the first unmarked 0 by scanning the tape and mark it. If there are no unmarked 0, go to step 4. Else, move the head back to the front of the tape.

**Step 2:**

Mark the first unmarked 1 by scanning the tape and mark it. If there is no unmarked 1, then reject.

**Step 3:**

Place the head back to the front of the tape and repeat step 1.

**Step 4:**

Place the head back to the front of the tape and check if any unmarked 0's or 1's remain by scanning the tape. If there are none, accept else it is unacceptable.

Comments (1)

**Step 2** of 3

b.

Given language $L_2$ is { $w \mid w$ Contains twice as many 0's as 1's} over the alphabet $\{0,1\}$.

Let $M_2$ be the machine that decides the language L. The implementation level description of $M_2$ is as follows:

$M_2 =$" on input w:

**Step 1:**

Scan the tape for the first unmarked 1, mark it. If there are no unmarked 1's, and go to step 5. Else place the head at the start of the tape.

**Step 2:**

Scan the unmarked 0 is found in the tape and mark it. If there are no 0's, reject.

**Step 3:**

Scan the tape again till the unmarked 0's is said to be found and mark them. If there is no unmarked 0's, reject.

**Step 4:**

Move the head back to the front of the tape and repeat step 1.

**Step 5:**

Place the head back to the front of the tape scan the tape to see if there are any unmarked 0's are found. If none are found, then accept. Else, reject.

Comments (1)

c.

Given language $L_3$ is { $w \mid w$ does not contain twice as many 0's as 1's} over the alphabet $\{0,1\}$.

Let $M_3$ be the machine that decides the language L. The implementation level description of $M_3$ is as follows.

$M_3 =$" on input w:

**Step 1:**

Scan the tape and mark the first unmarked 1, mark it. If there is no unmarked 1's, go to stage 5. Else, place the head at the start of the tape.

**Step 2:**

Scan the tape till the unmarked 0's are found and mark it. If there are no 0's accept.

**Step 3:**

Scan the tape once again and marks the unmarked 0's are found. If there are no 0's, accept.

**Step 4:**

Place the head back to the front of the tape and repeat step 1.

**Step 5:**

Place the head back to the front of the tape. Scan the tape to see if there are unmarked 0's. If there are none, reject. Else, accept.

Comments (3)

## Problem

Let a k-PDA be a pushdown automaton that has k stacks. Thus a 0-PDA is an NFA and a 1-PDA is a conventional PDA. You already know that 1-PDAs are more powerful (recognize a larger class of languages) than 0-PDAs.

**a.** Show that 2-PDAs are more powerful than 1-PDAs.

**b.** Show that 3-PDAs are not more powerful than 2-PDAs.

(Hint: Simulate a Turing machine tape with two stacks.)

## Step-by-step solution

**Step 1** of 3

**Push down automata:**

Push down automata (PDA) is like nondeterministic finite automata but have an extra component called stack. Stack provides additional memory beyond the finite amount available in the control.

• *k*-PDA is a push down automaton that has $k-$stacks

• $0-\text{PDA}$ is a NFA

• $1-\text{PDA}$ is a conventional $\text{PDA}$.

It is known that $1-\text{PDAs}$ are more powerful than $0-\text{PDAs}$.

It means $1-\text{PDAs}$ are recognized by a larger class of languages than $0-\text{PDAs}$.

Comment

**Step 2** of 3

(a)

Now it is required to show that $2-\text{PDAs}$ are more powerful than $1-\text{PDAs}$.

Prove this by taking an example.

As it is known that "A Language is context free if and only if some pushdown automaton recognizes it."

Let $B=\left\{a^n b^n c^n \mid n \geq 0\right\}$ be the language.

*P* is the pumping length of the B given by the pumping lemma.

Now prove that B is not a CFL (Context-Free Language).

To show that B is not a CFL, it's enough to show that a string $s=1^p 0^p 1^p 0^p$ cannot be pumped.

Consider *s* is of the form *s=uvxyz.*

• If both *v* and *y* contain at most one type of alphabet symbol, the string will be of the form $uv^2xy^2z$ runs of $0\text{'s and } 1\text{'s}$ of unequal length. Hence the string *s* cannot be a member of B.

• If either *v* or *y* contains more than one type of alphabet symbol, the string will be of the form $uv^2xy^2z$ which does not contain the symbols in correct order. Hence the string *s* cannot be a member of B.

Since the string *s* cannot be pumped without violating the pumping lemma condition, *B* is not a CFL (context-free language). Thus $1-\text{PDA}$ does not recognize *B*.

But the following $2-\text{PDA}$ recognizes *B*.

• Push all $a's$ that appear in front of the input tape to stack 1.

• Then push all $b's$ that follow the $a's$ in the input stream into stack2. If it sees any $a's$ at this stage, ***reject.***

• When it sees the first *c*, pop stack 1 and stack2 at the same time. After this, reject the input if it sees any $a's$ (or) $b's$ in the input stream.

• Pop stack1 and stack2 for each input character *c* it reads. If the input ends when both stacks are empty, ***accept***. Otherwise ***reject.***

Therefore, B is recognized by $2-\text{PDA}$.

Thus $2\text{PDA}s$ are more powerful then $1-\text{PDA}s$.

Comments (2)

**Step 3** of 3

(b)

To show that 3-PDAs are not more powerful than $2\text{PDA}s$,

It is known that 3-PDA is not powerful than Turing machine, so it is required to show that the $2-\text{PDA}$ can simulate a Turing machine, then obviously 3-PDAs are not powerful than 2-PDAs.

**Simulation of TM by 2PDA:**

Now split the tape of TM (Turing machine) into two stacks.

• Stack1 stores the characters on the left of the head, with the bottom of stack storing the left most character of tape in the TM.

• Stack 2 stores the characters on the right of the head, with the bottom of the stack storing the right most character on the tape in the TM.

Show the Transition example for left and right.

Left Transition:

• If $\delta(q_i, w_i) = (q_j, w_j, L)$, then $w_i$ on top of second stack.

• If $ is on top of stack then reject. Move left edge of tape.

o Otherwise pop $w_i$ from top of second stack. Push $w_j$ in second stack.

• Pop the symbol of first stack and push it to second stack. Switching the state of machine $q_j$.

• If $q_j$ is accept state, then accepts *w*. If $q_j$ is reject state, then rejects *w.*

Right Transition:

• If $\delta(q_i, w_i) = (q_j, w_j, R)$, then $w_i$ on top of second stack.

• Pop $w_i$ from second stack and push $w_j$ in second stack

• If $ is on top of second stack, push null on to second stack. Either case, switching the state of machine $q_j$.

• If $q_j$ is accept state, then accepts *w*. If $q_j$ is reject state, then rejects *w.*

In this way $2-\text{PDA}$ simulates the Turing Machine.

Furthermore,

Now simulate 4-tape non-deterministic Turing machine with 3-PDA.

• First tape as input tape, second, third, and fourth tapes respectively first, second and third stack.

• If stack changed by push without pop then N moves the tape head to right without change the current symbol. Now write the symbol to the tape which is pushed on stack.

• If stack is changed by pop without push, then N replaces tape head with blank symbol and moves left.

• If stack is changed by push and pop, then N changes the tape by written symbol pushed in current cell.

• If stack is not changed, then N writes same symbol on tape head and then back to tape.

• Symbol is used from the tape then N moves first tape to right.

• If symbol is not used then N copies the element at first tape head.

• These actions can parallel in all tapes at same time since they take only one turn.

• 2$^{nd}$ point mentioned Push without pop, in this case all tape heads kept same except stacks affected by push without pop. Moves right to get the blank symbol at tape.

• N is nondeterministic to guess which transition to take same way PDA does.

So, it proves that Turing Machine are powerful as 3-PDAs, so it can be concluded that 3-PDAs are no more powerful than 2-PDAs. Both the PDAs are equally powerful.

Comments (1)

## Problem

Say that a ***write-once Turing machine*** is a single-tape TM that can alter each tape square at most once (including the input portion of the tape). Show that this variant Turing machine model is equivalent to the ordinary Turing machine model. (Hint: As a first step, consider the case whereby the Turing machine may alter each tape square at most twice. Use lots of tape.)

## Step-by-step solution

**Step 1** of 2

To show that write – once Turing machine is equivalent to an ordinary Turing machine, first we simulate an ordinary Turing machine by a write – twice Turing machine.

**Simulation of an ordinary Turing machine by a write – twice Turing machine:**

Let the original machine be M. The two techniques by which it can be shown that write-once Turing machine is equivalent to an ordinary Turing machine.

The first technique is to copy the entire tape between transitions. In this one extra tape symbol is required as a delimiter and an expanded alphabet is required to record the position of the tape as it is copied.

In this case two writes are required: The first write is used while copying to record the symbol over fresh tape, the second write is used to mark a symbol as copied.

Comment

**Step 2** of 2

The second technique is that the use of tape is not required. In this two tape squares are required to be used, instead of recording each symbol in one tape square. The first square is used to record a tape symbol and the second is used to mark the symbol as being copied. Each tape square can be changed only once.

Comment

## Problem

A ***Turing machine with doubly infinite tape*** is similar to an ordinary Turing machine, but its tape is infinite to the left as well as to the right. The tape is initially filled with blanks except for the portion that contains the input. Computation is defined as usual except that the head never encounters an end to the tape as it moves leftward. Show that this type of Turing machine recognizes the class of Turing-recognizable languages.

## Step-by-step solution

**Step 1** of 2

To prove the double infinite tape Turing machine *D* is equals to Normal ordinary Turing machine *M.*

In order to show the equivalence between *D* and *M*, it is necessary show the following two things:

• First one is, any language *L* that can be recognized by *M* can be recognized by *D* as well, and

• Second part is, any Language *L'* that can be recognized by *D* can be recognized by *M.*

This can be showed by simulation. That is, simulate *D* to act like *M* and vice-versa.

(i) **Simulating *M* by *D*:**

**Step 1:** Mark the left hand end of *D.*

**Step 2:** Prevent *D* from moving its head to the left of the mark.

In this way *D* simulates *M.*

Comment

**Step 2** of 2

(ii) **Simulating *D* by *M*:**

To simulate the doubly infinite tape TM(Turing machine) *D* by an ordinary TM *M,* simulate it with a 2-tape TM.

As the 2 tape TM was already equivalent is power to an ordinary TM.

**Simulation of doubly infinite tape TM ' *D* ' with 2-tape TM ' $M_1$ ':**

The first tape of $M_1$ is written with input string and the second tape is blank.

Now cut the tape of doubly infinite tape Turing machine *D* into two parts, at the starting cell of the input string.

• One part containing input string and all the blank spaces to its right.

• Second part containing left of the input string appears on the second tape, in reverse order.

In this way *M* simulates *D.*

Thus from (i) and (ii)

*D* is equivalent to *M* i.e., Turing machine with a doubly infinite tape is equivalent to an ordinary Turing machine.

Comment

## Problem

A ***Turing machine with left reset*** is similar to an ordinary Turing machine, but the transition function has the form

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{R}, \mathrm{RESET}\}.$$

If ? (q, a) = (r, b,RESET), when the machine is in state q reading an a, the machine's head jumps to the left-hand end of the tape after it writes b on the tape and enters state r. Note that these machines do not have the usual ability to move the head one symbol left. Show that Turing machines with left reset recognize the class of Turing-recognizable languages.

## Step-by-step solution

**Step 1** of 2

**To show the Turing machine with left reset, recognize the class of Turing recognizable class of language.**

• It must be shown that it can simulate ordinary Turing machine.

• Let *M* be an ordinary Turing machine and $M_L$ be the Turing machine with left reset.

• $M_L$ simulates *M* in the following way.

• When *M* makes a right transition then $M_L$ follows it in the same way as *M* do.

• When *M* makes a left transition with symbol *a*, *b* in *M*, $M_L$ replaces it with *A* or *B* respectively. So, the alphabet set $\sum M_L = \sum MU\{A,B\}$ and does a left RESET.

• Shifts all content of the tape by one position to the right for all symbols other than $\{A,B\}$.

Comment

**Step 2** of 2

The above process is repeated until all content of the tape are shifted to the right and does the following.

• $M_L$ does a RESET again.

• All right transitions are checked.

• Whenever it reaches to some $\{A,B\}$, it works in the same way as *M* does.

Comment

## Problem

A ***Turing machine with stay put instead of left*** is similar to an ordinary Turing machine, but the transition function has the form

$$\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{R}, \mathrm{S}\}.$$

At each point, the machine can move its head right or let it stay in the same position. Show that this Turing machine variant is *not* equivalent to the usual version. What class of languages do these machines recognize?

## Step-by-step solution

**Step 1** of 2

A Deterministic Finite State automaton can be simulated on the Turing Machine with stay put instead of left. The modifications can be done if transitions are added from state in F to $q_{accept}$ and from the states outside F to $q_{reject}$ when a blank symbol is read.

Assume there is a Turing Machine M, such that $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ with stay put instead of left. Create a DFA such that the DFA $(Q', \Sigma', \delta', q_0', F)$ recognizes the same language.

The machine M cannot move left and cannot write anything that it can written on the tape while moving to the right. Thus, the access is one-way.

For every DFA, there exists a Turing Machine that accepts the same language because a DFA is a Turing Machine with read only tape and tape head with moves to right.

Comments (2)

**Step 2** of 2

The transition function $\delta'$ for the NFA is as follows:

First, set $\delta'(q_{\text{start}}, P) = \{q_{OP}\}$ where $q_{op}$ is the start state of TM variant.

Next, set $\delta'(q_{accept}, i) = \{q_{\text{accept}}\}$ For any *i*

If $\delta(p,a) = (q_{\text{accept,}} b, w)$ where $w = R$ or $S$, set $\delta'(q_{pa}, \in) = \{q_{\text{acceept}}\}$

R is RIGHT *S* is stay put.

If $\delta(p,a) = (q_{reject}, b, w)$ where $w = R$ or $S$, we set $\delta'(q_{pa} \in) = \{q_{reject}\}$

• For each $a \in \Sigma$, set $\delta'(q_{start}, a) = \{\langle q_0, a\rangle\}$, where $q_0$ is start state of S.

• For each $p, q \in Q$ where $p \notin \{q_{accept}, q_{reject}\}$, for each $a \in \Gamma$, if S has transition of form $\delta(p,a) = (q_{\text{accept,}} b, w)$ or $\delta(p,a) = (q_{reject}, b, w)$, *w* becomes R for each $c \in \Sigma$, set $\delta'(\langle p,a\rangle, c) = \{\langle q, c\rangle\}$.

• For each $p, q \in Q$ where $p \notin \{q_{accept}, q_{reject}\}$, for each $a \in \Sigma$, if S has transition of form $\delta(p,a) = (q_{\text{accept,}} b, w)$ or $\delta(p,a) = (q_{reject}, b, w)$, *w* becomes S then set $\delta'(\langle p,a\rangle, \varepsilon) = \{\langle q, b\rangle\}$

Thus, an NFA is constructed which is defined as follows:

$(Q' = Q, \Sigma' = \Sigma, \delta', q_{op} = q_{start}, F)$ From our TM variant *S*.

The language recognized by NFA is regular languages.

Comment

## Problem

A ***queue automaton*** is like a push-down automaton except that the stack is replaced by a queue. A ***queue*** is a tape allowing symbols to be written only on the left-hand end and read only at the right-hand end. Each write operation (we'll call it a *push*) adds a symbol to the left-hand end of the queue and each read operation (we'll call it a *pull*) reads and removes a symbol at the right-hand end. As with a PDA, the input is placed on a separate read-only input tape, and the head on the input tape can move only from left to right. The input tape contains a cell with a blank symbol following the input, so that the end of the input can be detected. A queue automaton accepts its input by entering a special accept state at any time. Show that a language can be recognized by a deterministic queue automaton iff the language is Turing-recognizable.

## Step-by-step solution

**Step 1** of 3

The equivalence between the queue automaton Q and the Turing machine M is required to be shown.

It means it is required to be shown that the language that can be recognized by the automaton, it can also be recognized by the Turing machine and vice versa.

This can be done by showing simulation which means that simulate the automaton Q to behave exactly like the Turing machine M and vice versa.

Comment

**Step 2** of 3

The Automaton can be simulated by the Turing machine as follows:

Consider the entire tape as a queue. One by one each symbol is altered and the movement of the tape takes place to the right. If more than one numbers are to be pushed in the queue, then it is done by shifting contents to the right. If the end of the tape is reached, the leftmost symbol of the tape is approached.

Comment

**Step 3** of 3

The Turing machine can be simulated by the automaton as follows:

The alphabet of the Turing machine M is expanded by inserting an extra symbol. A left end marker # is inserted to the queue. The symbols are pushed to left and read (popped) from the right.

Comment

## Problem

Show that the collection of decidable languages is closed under the operation of

A**a.** union.

**b.** concatenation.

**c.** star.

**d.** complementation.

**e.** intersection.

## Step-by-step solution

**Step 1** of 6

A language is said to be decidable if and only if some Turing machine decides it. The Turing machine is a decider if all braches halts on all inputs.

Comments (2)

**Step 2** of 6

a.

Let $L_1$ and $L_2$ be two decidable Languages. $M_1$ and $M_2$ be the Turing machines that decides $L_1$ and $L_2$ respectively.

There exists a Turing machine $M'$ such that, decides $L_1 \cup L_2$ i.e. $L(M') = L_1 \cup L_2$.

The description of $M'$ is as follows:

$M' =$on input w:

1. Run $M_1$ on *w*. if $M_1$ accepts, then ***accept***.

2. Else Run $M_2$ on *w*. If $M_2$ accepts, then ***accept***

3. Else ***reject***

$M'$ Accepts *w* if either $M_1$ or $M_2$ accepts it. If both rejects, $M'$ rejects.

**Therefore,** $L(M') = L_1 \cup L_2$ **. The decidable languages are closed under union.**

Comment

**Step 3** of 6

b.

Let $L_1$ and $L_2$ be two decidable Languages. $M_1$ and $M_2$ be the Turing machines that decides $L_1$ and $L_2$ respectively.

There is a Turing machine $M'$ such that, it decides concatenation of $L_1$ and $L_2$i.e., $L(M') = L_1 \circ L_2$.

The description of $M'$ is as follows:

$M' =$on input *w*:

1. Split *w* into two parts $w_1, w_2$ such that $w = w_1 w_2$

2. Run $M_1$ on $w_1$. If $M_1$ rejected then ***reject***.

3. Else run $M_2$ on $w_2$. If $M_2$ rejected then ***reject***.

4. Else ***accept***

Try each possible cut of *w*. If first part is accepted by $M_1$ and the second part is accepted by $M_2$ then *w* is accepted by $M'$. Else, *w* does not belong to the concatenation of languages and is rejected.

**Therefore,** $L(M') = L_1 o L_2$ **. The decidable languages are closed under concatenation.**

Comment

**Step 4** of 6

c.

Let *L* be a Turing decidable Language and *M* be the Turing machine that decides *L*.

There is a Turing machine $M'$ such that, it decides star of *L* i.e., *L*( $M'$)=$L^*$.

The description of $M'$ is as follows:

$M' =$ On input *w*:

1. Split *w* into *n* parts such that

$w = w_1 w_2 ... w_n$ in different ways.

2. Run *M* on $w_i$ for $i = 1, 2, ... n$.

If *M* accepts each of these strings $w_i$, ***accept***.

3. All cuts have been tried without success then ***reject***.

When *w* is cut into different substrings such that every string is accepted by *M*, then *w* belongs to the star of *L* and thus $M'$ accepts *w* after finite number of steps, else *w* will be rejected. Since, there are finitely many possible cuts of *w*, $M'$ will halt after finitely many steps.

**Therefore,** $L(M') = L^*$ **. The decidable languages are closed under star.**

Comment

**Step 5** of 6

d.

For a Turing decidable language L, Turing machine decides language *M* then the complement is $M'$ on input *w*.

The description of $M'$ is as follows:

$M' =$on input *w*:

1. Accepts if *M* rejects

2. Else ***accept.***

Since $M'$ does the opposite of what ever $M$ does, it decides the complement of $L$.

**Therefore, decidable languages are closed under complementation.**

Comments (3)

**Step 6** of 6

e.

Let $L_1$ and $L_2$ be two Turing decidable Languages. $M_1$ and $M_2$ be the Turing machines that decides $L_1$ and $L_2$ respectively.

There is a Turing machine $M'$ such that, it decides intersection of $L_1$ and $L_2$ i.e., $L(M') = L_1 \cap L_2$.

The description of $M'$ is as follows:

$M' =$on input *w*:

1. Run $M_1$ on *w*. if $M_1$ rejects then ***reject***.

2. Else run $M_2$ on *w*. if $M_2$ rejects then ***reject***.

3. Else ***accept***.

$M'$ Accepts *w* if both $M_1$ and $M_2$ accept it. If either of them rejects then $M'$ rejects *w*.

**Therefore,** $L(M') = L_1 \cap L_2$ **. The decidable languages are closed under intersection.**

Comments (2)

## Problem

Show that the collection of Turing-recognizable languages is closed under the operation of

A**a.** union.

**b.** concatenation.

**c.** star.

**d.** intersection.

**e.** homomorphism.

## Step-by-step solution

**Step 1** of 8

• Suppose, $X$ and $Y$ be two Turing recognizable languages that have Turing machines $M_X$ and $M_Y$ respectively.

• Now the union of these languages is denoted by $L_{XY}$ and the Turing machine recognizing this language is $M_{XY.}$

"On input *w:"*

1. Run X and Y alternately on $w$ step by step. If either accepts, *accept*. If both halt and reject, *reject.*

Suppose $s$ be a word from $L_{XY}$. $M_{XY}$ works for an input string $s$ as shown:

• It executes $M_X$ and $M_Y$ on $s$ individually.

• If at least any one of $M_X$ or$M_Y$ accepts $s$ then $M_{XY}$ also accepts after a finite number of steps and reach to its accepting state.

• If both $M_X$ and $M_Y$ reject and either of them do so by looping then $M_{XY}$ will loop.

Comment

**Step 2** of 8

**Hence, it can be said that collection of Turing recognizable languages is closed under union operation.**

Comment

**Step 3** of 8

• Suppose, $X$ and $Y$ be two Turing recognizable languages that have Turing machines $M_X$ and $M_Y$ respectively. • Now the concatenation of these languages is denoted by $L_{XY}$ and the Turing machine recognizing this language is $M_{XY.}$

Let $s$ be a word from $L_{XY}$. $M_{XY}$ works for an input string $s$ as shown:

• It divides each string of $XY$ into $s_1$ and $s_2$ non-deterministically.

• It runs $s_1$ to $M_X$. If $M_X$ halts and rejects, *reject*.

• If runs $s_2$ to $M_Y$. If $M_Y$ accepts, *accepts*. If $M_Y$ halts and rejects, *reject*.

Comments (2)

**Step 4** of 8

**Hence, it can be said that collection of Turing recognizable languages is closed under concatenation operation.**

Comment

**Step 5** of 8

• Suppose, $X$ be a Turing recognizable language and the Turing machine is $M_X$.

• Now $X^*$ is the language obtained from star operation on $X$.

The Turing machine for this language be $M_{X^*}$.

$M_{X^*}$ works as follows:

• For an input string $s$ of $X$, it non-deterministically divided the string into $s_1, s_2 \ldots s_n$.

• For each of those divided parts $M_{X^*}$ runs, Suppose, $M_{X^*}$ all divided parts then $s$ is accepted by $M_{X^*}$ else $s$ is rejected by $M_{X^*}$.

Comment

**Step 6** of 8

**Hence, it can be said that collection of Turing recognizable languages is closed under star operation.**

Comment

**Step 7** of 8

• Suppose, $X$, $Y$ be two Turing recognizable languages that have Turing machines $M_X$ and $M_Y$ respectively.

• Now the intersection of these languages is denoted by $L_{XY}$ and the Turing machine recognizing this language is $M_{XY}$.

For an input string $s$ from $L_{XY}$, $M_{XY}$ works for as shown:

• Turing machine $M_X$ runs on s. If it accepts s then $M_Y$ runs on $s$. Else $s$ is rejected.

• Suppose, $M_Y$ accepts $s$ then it is accepted by the Turing machine otherwise $s$ is rejected.

**Hence, it can be said that collection of Turing recognizable languages is closed under intersection operation.**

Comments (5)

**Step 8** of 8

• Suppose, $X$ a Turing recognizable language that have Turing machine $M_X$.

• To recognize $h(X)$ the other Turing machine $M_Y$ is simulated in such a way that:

On input $s$, it will consider all strings $w$ such that $h(w) = s$.

• The *TM* $M_X$ will execute on input $w$ by going through all strings in $w$.

If $h(w) = s$ start executing $M_X$ on input $w$, using merging to interleave with other executions on $M_X$. *Accept* if any executions accept.

• $M_Y$ will accept $s$ it any of those executions of $M_X$ accepts $s$. Else s will be rejected.

**Hence, it can be said that collection of Turing recognizable languages is closed under homomorphism operation.**

Comment

## Problem

Let $B = \{\langle M_1 \rangle, \langle M_2 \rangle, \ldots\}$ be a Turing-recognizable language consisting of TM descriptions. Show that there is a decidable language C consisting of TM descriptions such that every machine described in B has an equivalent machine in C and vice versa.

## Step-by-step solution

**Step 1** of 2

In order to solve this problem, we need to know the definition of enumerator and some theorems

**Enumerator:-**

An enumerator is a Turing machine that consist a work tape and the output tape. It outputs the strings by using the work tape without accepting any input.

Also we use the following theorem

**Theorem 1:**

"A language is Turing – decidable if and only if some enumerator enumerates the strings of this language in lexicographic order"

Comment

**Step 2** of 2

Consider the language $B = \{\langle M_1 \rangle, \langle M_2 \rangle \ldots\}$.

*B* is a Turing recognizable language.

*C* is a language consisting of Turing machines descriptions.

Consider *E* be the enumerator for the Turing recognizable language *B*.

Construct an enumerator $E_o$ which output the strings of *C* in lexicographic order.

From the above Theorem1, *C* is decidable.

Enumerator $E_o$ simulates *E*.

When *E* gives the i$^{th}$ TM $\langle M_i \rangle$ as output, then enumerator $E_o$ pads $M_i$ by adding sufficiently many extra useless states to obtain a new TM $M_i'$ where the length of $\langle M_i' \rangle$ is greater than the length of $\langle M_{i-1}' \rangle$. Then E outputs $\langle M_i' \rangle$.

Thus simulation occurs in both directions.

Therefore, $E_o$ and *E* are equivalent.

Comments (1)

## Problem

Show that a language is decidable iff some enumerator enumerates the language in the standard string order.

## Step-by-step solution

**Step 1** of 3

To prove this problem:

We need to show equivalence between a Turing machine that decides a language and an enumerator that enumerates it. Hence we have to show the proof in both directions i.e.,

1. Language is decidable then enumerator enumerates the language in lexicographic order.

2. Enumerator enumerates the language in lexicographic order and then it is decidable.

Comment

**Step 2** of 3

1. Language is decidable then enumerator enumerates the language in lexicographic order.

**Proof:**

**<u>If direction:</u>**

Let us assume that we have a Turing machine $M$ to decide a language $L$.

Now we can use this $M$ to construct an enumerator $E$ as follows.

We generate the strings in the lexicographic order and input each string into $M$ for $L$.

If $M$ accepts then print that string. Therefore $E$ prints all strings of $L$ in lexicographic order.

Comment

**Step 3** of 3

2. Enumerator enumerates the language in lexicographic order and then it is decidable.

**Proof:**

**<u>And if direction: -</u>**

Now we need to consider the other direction.

That is, if we have an enumerator $E$ for a language $L$, then we can use $E$ to construct a Turing machine $M$ that decides $L$.

Here we have to consider two cases.

**<u>Case (i): If L is finite:</u>**

If L is finite language, then it is decidable, because all finite languages are decidable.

**<u>Case(ii) If L is infinite:-</u>**

If L is infinite then a decider for $L$ operates as follows.

• On receiving the input $w$, the decider enumerates all strings of $L$ in lexicographic order until a string greater than $w$ appears in the lexicographic order.

• This must eventually occur since $L$ is infinite.

• If $w$ has appeared in the enumeration already, then *accept*.

• If $w$ has not yet appeared in the enumeration then it will never appear and hence we can *reject*.

So in both cases $L$ is decidable. The theorem proved in both directions.

Therefore,

**A language is decidable if and only if some enumerator enumerates the language in lexicographic order.**

Comment

## Problem

Show that every infinite Turing-recognizable language has an infinite decidable subset.

## Step-by-step solution

**Step 1** of 2

Let *L* be an infinite Turing recognizable language.

We know that

"A language is Turing recognizable if and only if some enumerator enumerates it"

So, let M be the enumerator that enumerates L.

Consider a language $L' = \{w_1, w_2, w_3 ....\}$ where

$\rightarrow w_1$ is the first string enumerated by *M*

$\rightarrow$ For every $i > 1$,

$w_i$ is the first string enumerated by *M* and that is lexicographically larger than $w_{i-1}$.

(i) **<u>First we prove $L'$ is infinite and is subset of L:</u>**

Let us assume that $L'$ is finite

Then $w_i$ is the last and lexicographically largest element in $L'$ and all strings enumerated by M must be lexicographically less than $w_i$.

Since there are only a finite number of strings that are less than $w_i$, L would then be finite.

This is a contradiction because *L* is infinite.

Therefore, our assumption that $L'$ is finite is wrong.

Hence $L'$ is infinite.

All the strings in $L'$ were at some point enumerated by M.

So clearly $L'$ is subset of *L*.

Comment

**Step 2** of 2

(ii) **<u>Next we have to prove $L'$ is decidable</u>:**

Now we will show that $L'$ is decidable for the given enumerator *M*, we can construct an enumerator $M'$ in lexicographic order.

$M'$ does the following

• Let *w* be the last string that $M'$ emitted, and initialize *w* to a dummy value that comes lexicographically before all strings.

• Simulate *M* until it emits a string *t*

• If: $t > w$ lexicographically

Then set $w = t$ and Let $M'$ emits *t*.

• Else: Ignore *t*.

• Resume simulating *M*, and go to step 2

Thus $M'$ is constructed and that enumerated $L'$ in lexicographic order.

We know that,

"A language is decidable if and only if some enumerator enumerates the language in lexicographic order"

So by this theorem $L'$ is decidable.

Therefore from (i) and (ii) $L'$ is an infinite decidable subset of *L*.

Comment

## Problem

Show that single-tape TMs that cannot write on the portion of the tape containing the input string recognize only regular languages.

## Step-by-step solution

**Step 1** of 3

Let $M$ be the single – tape Turing machine that cannot write on the portion of the tape containing the input string.

$M = (Q, \Sigma, \tau, q_0, q_{accept}, q_{reject})$

$M$ works on an input string *x* as follows.

Here we consider two events.

(i) **Out event:**

In out event, the tape head moves from input portion to non – input portion, i.e., the portion of the tape on the right of the $(x)^{th}$ cell.

(ii) **In event:**

In In-event tape head moves from non – input portion to input portion.

Comment

**Step 2** of 3

Consider the state $q_x$ for Turing machine $M = (Q, \Sigma, \tau, q_0, q_{accept}, q_{reject})$ when it first enters the non – input portion (i.e., after it's first *out event*)

• In case $M$ never enters the non – input portion.

(a) If $M$ accepts *x,* assign $q_x = q_{accept}$

(b) If $M$ does not accept *x,* assign $q_x = q_{reject}$

For any $q \in Q$, define a characteristic function $f_x$ such that

$f_x(q) = q'$

That implies

If *M* is in the state *q* and about to perform an "*in event*", the next "*out event*" will change M in state $q'$

• In case $M$ never enters the non – input portion again,

(a) If $M$ enters the accept state inside the input portion, assign $f_x(q) = q_{accept}$

(b) If $M$ does not enter the accept state, assign $f_x(q) = q_{reject}$

For two strings *x* and *y*,

If $q_x = q_y$ for all *q*, $f_x(q) = f_y(q)$, then *x* and *y* are indistinguishable by $M$. That is, $M$ accepts *xz* if and only if $M$ accepts *yz*.

Comments (1)

**Step 3** of 3

As there are finite choice of $q_x$ and $f_x$ $\left(\text{Precisely } |Q|^{|Q|+1} \text{ such choices}\right)$, the number of indistinguishable strings are finite.

"Myhill – Nerode theorem" is used to prove whether the language is regular or not.

**Statement:**

A language L over alphabet $\Sigma$ is regular if and only if the set of equivalent classes of $I_L$ is finite.

$I_L$ is the relation on $\Sigma^*$ such that for two strings *x* and *y* of $\Sigma^*$

$$x I_L y \Leftrightarrow \{z \mid xz \in L\} = \{z \mid yz \in L\}$$

That is $xI_L y$ if and only if they are indistinguishable with respect to L

So, by Myhill – Nerode theorem the language recognized by $M$ is regular.

Comments (1)

## Problem

Let $c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1}$ be a polynomial with a root at x = $x_0$. Let $c_{max}$ be the largest absolute value of a ci. Show that

$$|x_0| < (n+1)\frac{c_{\max}}{|c_1|}.$$

## Step-by-step solution

**Step 1** of 3

Consider a polynomial $f(x) = c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1}$.

Since above polynomial has a root at

$x = x_0$

So $f(x_0) = 0$ implies that

$c_1x_0^{\ n} + c_2x_0^{\ n-1} + \cdots + c_nx_0 + c_{n+1} = 0$

Rearrange the above equation as:

$c_1x_0^n = -\left(c_2x_0^{n-1} + \ldots + c_nx_0 + c_{n+1}\right)$

Now, take modulus on both the sides of the equation as:

$$\left|c_1x_0^{\ n}\right| = \left|-\left(c_2x_0^{\ n-1} + \cdots + c_nx_0 + c_{n+1}\right)\right|$$
$$\left|c_1\right|x_0^{\ n} = \left|\left(c_2x_0^{\ n-1} + \cdots + c_nx_0 + c_{n+1}\right)\right|$$

Comments (4)

**Step 2** of 3

Use sub-additive property $|a+b| \le |a| + |b|$ of modulus function,

$\left|c_1x_0^{\ n}\right| \le \left|c_2x_0^{\ n-1}\right| + \ldots + \left|c_nx_0\right| + \left|c_{n+1}\right|$

Also, as $c_{\max}$ is the largest absolute value of $c_i$ then for each $i = 1, 2, \cdots, (n+1)$,

$c_{\max} = \left|c_{n+1}\right|$

So, from above equation,

$\left|c_1x_0^{\ n}\right| \le c_{\max}\left(1 + \left|x_0\right| + \ldots + \left|x_0^{\ n-1}\right|\right)$

Comments (2)

**Step 3** of 3

Substitute $n.x_0^{\ n-1}$ for $1 + \left|x_0\right| + \ldots + \left|x_0^{\ n-1}\right|$ where, $x_0^{\ n-1}$ is the largest one if $x_0 > 1$:

$$\left|c_1 x_0^{\,n}\right| \le c_{\max} \cdot n \left|x_0^{\,n-1}\right|$$

$$\left|\frac{x_0^{\,n}}{x_0^{\,n-1}}\right| \le n.\frac{c_{\max}}{|c_1|}$$

$$\left|x_0^{\,n-(n-1)}\right| \le n.\frac{c_{\max}}{|c_1|}$$

$$|x_0| \le n.\frac{|c_{\max}|}{|c_1|}$$

It also can be written as:

$$|x_0| \le n.\frac{c_{\max}}{|c_1|}$$

To make the term $n.\frac{c_{\max}}{|c_1|}$ strictly greater than $|x_0|$, we can re-write as:

$$|x_0| < (n+1)\frac{c_{\max}}{|c_1|} \quad (\text{since, } n < n+1 \text{ always holds})$$

Comment

## Problem

Let A be the language containing only the single string s, where

$$s = \begin{cases} 0 & \text{if life never will be found on Mars.} \\ 1 & \text{if life will be found on Mars someday.} \end{cases}$$

Is A decidable? Why or why not? For the purposes of this problem, assume that the question of whether life will be found on Mars has an unambiguous YES or NO answer.

## Step-by-step solution

**Step 1** of 1

**Decidable language:** A language is decidable if and only if some Turing machine decides it.

We call a Turing machine a decider if all branches halt on all inputs.

The given data is, A is the language containing only the single string $s$, where

$$s = \begin{cases} 0 & \text{if life never will be found on Mars} \\ 1 & \text{if life will be found on mars someday} \end{cases}$$

So, the language A may contain either 0 or 1 but not both

Thus $A = \{0\}$ (or) $A = \{1\}$

In both these cases $A$ are finite and string $s$ is fixed, so we know the same finite language is always decidable.

We are not able to determine whether

$A = \{0\}$ or $A = \{1\}$.

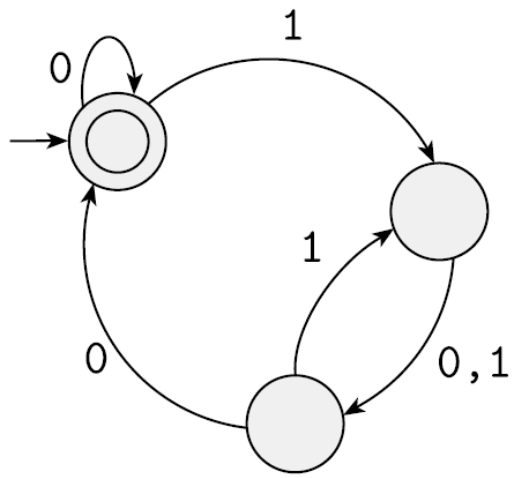
So we will not able to describe the decider for A.

In this case, we need to give two Turing machines for both the cases. So, definitely one of them will be the decider of *A*.

Comment

## Problem

Answer all parts for the following DFA M and give reasons for your answers.




**a.** Is $\langle M, 0100\rangle \in A_{\mathsf{DFA}}$?

**b.** Is $\langle M, 011\rangle \in A_{\mathsf{DFA}}$?

**c.** Is $\langle M\rangle \in A_{\mathsf{DFA}}$?

**d.** Is $\langle M, 0100\rangle \in A_{\mathsf{REX}}$?

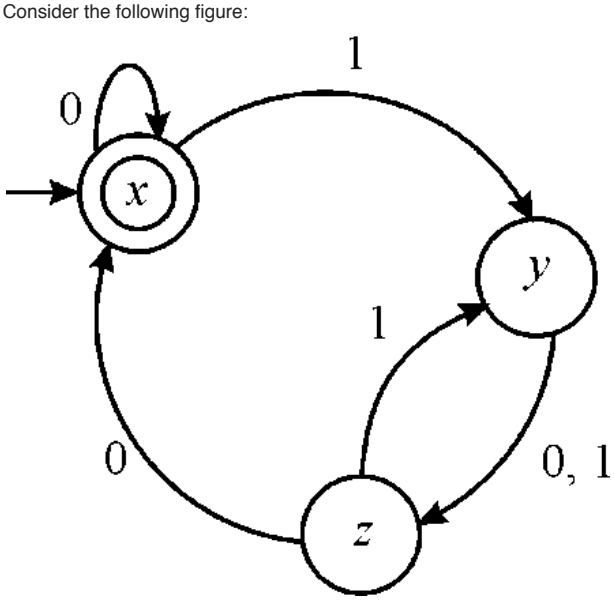
**e.** Is $\langle M\rangle \in E_{\mathsf{DFA}}$?

**f.** Is $\langle M, M\rangle \in EQ_{\mathsf{DFA}}$?

## Step-by-step solution

**Step 1** of 7

Consider the following figure:




As it is already given *M* is a DFA.

**Theorem:** $A_{DFA}$ is a language which test whether given DFA *M* accepts provided string *w* or not. Use the following steps to check DFA accepted given string or not.

Consider a Turing machine *W* which decides $A_{DFA}$.

$W = \text{On input } \langle M, w \rangle, \text{ where } M \text{ is a DFA and } w \text{ is string}$

1. On *w* input simulate *M*.

2. When simulation ends at accept states then it is **accepted.** When simulation does not end at accept states then it is **rejected**.

Comment

**Step 2** of 7

Use above theorem to check whether *M* accepts string $0100$ or not. Simulate, DFA *M* on $0100$. Consider the figure shown above for the production of *M*.

• As shown in above figure the starting and final (accepting) state of *M* is *x*. After taking the first input 0 from string $0100$ the next state is still *x*.

• After taking the second input 1 from string $0100$ the next state is *y*. similarly after taking fourth '0' input from $0100$ the next state becomes *x* which is an acceptable state.

Hence, *M* accepts input string $0100$.

Comment

**Step 3** of 7

Use above theorem to check whether *M* accepts string $011$ or not. Simulate, DFA *M* on $011$. Consider the figure shown above for the production of *M*.

• As shown in above figure the starting and final (accepting) state of *M* is *x*. After taking the first input 0 from string $011$ the next state is still *x*.

• After taking the second input 1 from string $011$ the next state is *y*. similarly after taking third '0' input from $011$ the next state becomes *z* which is not an acceptable state.

Hence, *M* does not accept input string $011$.

Comments (1)

**Step 4** of 7

Use above theorem to check whether *M* accepts given string or not. Here input string *w* is not given in $A_{DFA}$. Here $A_{DFA}$ is not in proper format.

Hence, *M* does not accept input string.

Comment

**Step 5** of 7

Consider a language $A_{REX}$ which is used to determine whether particular regular expression can generate provided string or not. Consider a Turing machine *W* which decides whether regular expression can generate provided string or not.

$W = \text{On input } \langle R, w \rangle, \text{ where R is a Regular expression and } w \text{ is string}$

Here, in this part *M* is a DFA instead of regular expression but for $A_{REX}$ it should be regular expression to check whether it can generate string $0100$ or not.

Hence, *M* cannot generate string $0100$.

Comment

**Step 6** of 7

Use marking algorithm $E_{DFA}$ to check, whether DFA accept any string or not. Use the following steps to check DFA accepted any string or not.

Consider a Turing machine *W* which decides $E_{DFA}$.

$W = \text{On input } \langle M \rangle \text{, where } M \text{ is a DFA}$

1. Marks the initial state of *M* hat is *x*.

2. Repeat the state till new state is not marked.

3. Mark the next state of transition which comes from any other marked states.

4. If accept state is not marked then **accept**, else **reject**.

Consider the transition diagram shown above. In that diagram the initial and final state are same that is *x*. so when applying the first step of $E_{DFA}$ theorem then initial as well as final states both get marked. According to step 4 of above theorem it is not acceptable when final states is marked.

Hence, *M* cannot accept any string. M is not an empty language.

Comments (5)

**Step 7** of 7

Consider theorem $EQ_{DFA}$ to determine whether two DFA identify the same language.

Consider a Turing machine *W* which decides $EQ_{DFA}$.

Here, in this part $L(M) = L(M)$ therefore derived language $L(C)$ becomes empty.

Therefore Turing machine W accepts the derived language $L(C)$.

Hence, yes the language of *M* accepts itself.

Comment

## Problem

Consider the problem of determining whether a DFA and a regular expression are equivalent. Express this problem as a language and show that it is decidable.

## Step-by-step solution

**Step 1** of 2

**Proof of the decidability of the language:**

• Express the language as $L = \langle R, S \rangle |$ R is a Deterministic Finite Automata(DFA) and S is a regular expression with $L(R) = L(S)$ )}.

• Recollect the Theorem 4.5 states a Turing machine *T* that decides the language $EQ_{DFA}$ = { $\langle P, Q \rangle$ | P and Q are Deterministic Finite Automata's(DFA) $L(P) = L(Q)$ }.

Comment

**Step 2** of 2

• Assume that *T* is the Turing Machine which decides language *L*.

• It can be defined as follows:

• T = "On input $L = \langle R, S \rangle$, where *R* is a Deterministic Finite Automata(DFA) and *S* is a regular expression:

• Convert R into a Deterministic Finite Automata(DFA) $D_R$ using the algorithm in the proof of Kleene's Theorem.

• Operate a Turing machine *TM* as a decider *F* using Theorem 4.5 on input $\langle R, D_S \rangle$.

• If *F* accepts, accept the language *L*.

• If *F* rejects, reject the language *L*.

Comments (2)

## Problem

Let $ALL_{\text{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \Sigma^*\}.$ Show that $ALL_{\text{DFA}}$ is decidable.

## Step-by-step solution

**Step 1** of 4

A DFA (Deterministic Finite Automaton) starts travelling, via arrows of the DFA, from the start state to the accept state and when it reaches an accept state, it accepts some string.

Comment

**Step 2** of 4

Consider the following details:

$ALL_{\text{DFA}} = \{(A) \mid A \text{ is a DFA and } L(A) = \Sigma^*\}$

Prove that $ALL_{\text{DFA}}$ is decidable.

Comment

**Step 3** of 4

*A* is a DFA that accepts every possible permutation and combination of its input string. Thus, its DFA has only a single state $q0$, which is both initial and final state.

Comments (3)

**Step 4** of 4

**So, on executing the Turing machine 'T' on INPUT( *A* ):**

• Mark the initial state of *A*.

• Repeat until no new states gets marked:

• The state that has any transition coming into it from any other already marked state will be marked.

• **ACCEPT**: when all the accept states are marked, otherwise **REJECT.**

Comments (4)

## Problem

Let

$$A\varepsilon_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG that generates } \varepsilon\}. \text{ Show that } A\varepsilon_{\text{CFG}}$$

is decidable.

## Step-by-step solution

**Step 1** of 1

**Decidability of the language**

**Given:** In this a language $A_{\varepsilon CFG}$ is given.

**Proof:** For showing that the language $A_{\varepsilon CFG}$ is decidable, build a Turing machine $T$ for deciding the language $A_{\varepsilon CFG}$. For all Context free grammars $G$

• If the grammar $G$ derives $\varepsilon$ then $T(\langle G\rangle)$ accepts

• If the grammar $G$ does not derive $\in$ then $T(\langle G\rangle)$ rejects.

**Constructions:**

For proofing the decidability of $A_{\varepsilon CFG}$ firstly convert the context free grammar $G$ into an equivalent $G'$ in CNF. If $S \to \varepsilon$ is the rule in the CFG G' then it means that $G'$ derives $\varepsilon$.

If the CFG $G'$ derives $\varepsilon$ then $G$ also derives it as $L(G) = L(G')$. As $G'$ is in CNF so only possible $\in$-rule in $G'$ is $S \to \varepsilon$. If $G'$ contains $S \to \varepsilon$ in production rules then $\varepsilon \in L(G')$. If $G'$ does not contains the rule $S \to \varepsilon$ then $\varepsilon \notin L(G')$.

Turing machine $T$ = on input $\langle G\rangle$ where $G$ is a context free grammar

• Convert the grammar $G$ in CFG $G'$.

• If $G'$ contains the production rule $S \to \varepsilon$ then accept it.

• Otherwise reject it.

**Conclusion:**

From the above construction it is clear that $\langle G\rangle \in A_{\varepsilon CFG}$ iff $\langle G, \varepsilon\rangle$ is also belongs to the $A_{CFG}$. So the above construction is correct. Hence the language $A_{\varepsilon CFG}$ is decidable.

Comment

## Problem

Let $E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a } \mathsf{TM} \text{ and } L(M) = \emptyset\}$. Show that $\overline{E_{\mathsf{TM}}}$, the complement of $E_{\mathsf{TM}}$, is Turing-recognizable.

## Step-by-step solution

**Step 1** of 1

**Re-cognition of the language by Turing machine**

**Given:** In this a language $E_{TM}$ is given. Show that the complement of $E_{TM}$ which is written as $\overline{E_{TM}}$ is recognized by Turing machine $M$.

**Proof:** Assume that $t_1, t_2, t_3, \ldots$ is a list of strings which are present in $\Sigma^*$. For proving $\overline{E_{TM}}$ is Turing recognizable, user has to determine whether any of the string from $t_1, t_2, t_3, \ldots$ is accepted by the Turing machine $M$ or not.

If the Turing machine $M$ accepts at least one string $t_i$ from the list $L(M) \neq \phi$ so the Turing machine $M$ belongs to $\overline{E_{TM}}$.

If the Turing machine $M$ does not accept any string then $L(M) = \phi$ so the Turing machine $M$ does not belong to $\overline{E_{TM}}$.

List of all strings cannot sequentially execute on the Turing machine $M$ as if the Turing machine $M$ can accept the string $\langle M\rangle \in \overline{E_{TM}}$ but it is looping in string $t_1$. Since the Turing machine $M$ accept the string $t_2$ from the list so $\langle M\rangle \in \overline{E_{TM}}$.

If list of all the strings is executed on $M$ in a sequential manner then user never extract the past of first string.

So for avoiding this problem related to sequential execution and recognizing $\overline{E_{TM}}$ construct a Turing machine.

**Construction of Turing machine M:**

Turing machine S= on input $\langle M\rangle$ here $M$ is a Turing machine.

1. Repeat the following methods for $i = 1, 2, 3, \ldots$
2. Execute each and every string from the list $t_1, t_2, t_3, \ldots$ on the Turing machine $M$.
3. If any string is accepted then accept it
4. Otherwise reject it.

**Conclusion:**

Hence the string is accepted by the Turing machine so the complement of $E_{TM}$ which is written as $\overline{E_{TM}}$ is Turing recognizable.

Comment

## Problem

Let X be the set {1, 2, 3, 4, 5} and Y be the set {6, 7, 8, 9, 10}. We describe the functions f : X → Y and g : X → Y in the following tables. Answer each part and give a reason for each negative answer.

| $n$ | $f(n)$ |
|---|---|
| 1 | 6 |
| 2 | 7 |
| 3 | 6 |
| 4 | 7 |
| 5 | 6 |

| $n$ | $g(n)$ |
|---|---|
| 1 | 10 |
| 2 | 9 |
| 3 | 8 |
| 4 | 7 |
| 5 | 6 |

A**a.** Is f one-to-one?

**b.** Is f onto?

**c.** Is f a correspondence?

A**d.** Is g one-to-one?

**e.** Is g onto?

**f.** Is g a correspondence?

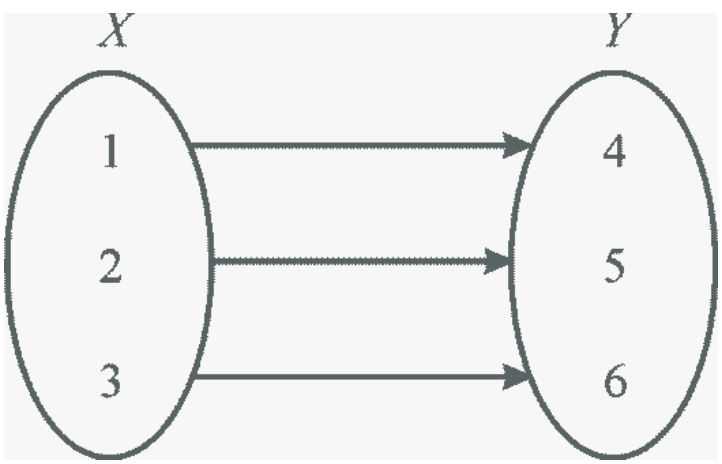
## Step-by-step solution

**Step 1** of 7

Given that X={1,2,3,4,5} and Y={6,7,8,9,10} .

a. **One-to-one Function:** A function $f: X \to Y$ is said to be one to one if and only if when for each and every member of $X$ there exist a unique matching member $Y$. It is also known as injective function.

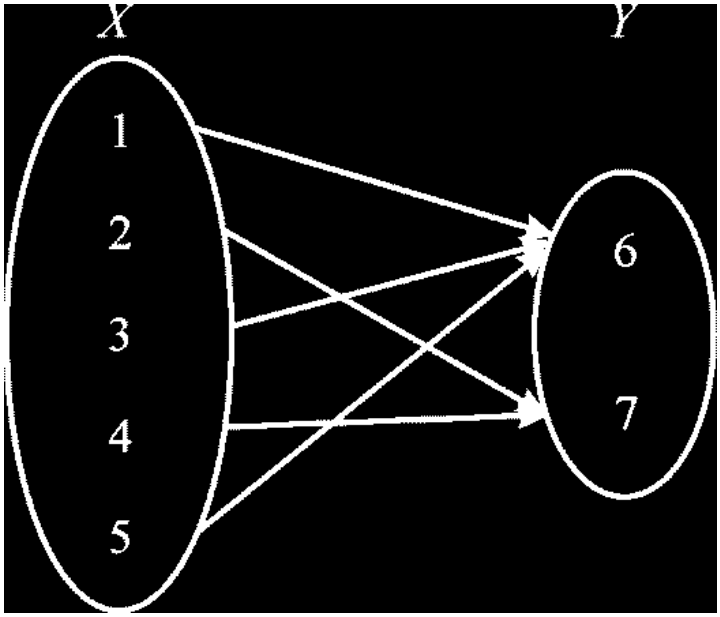
**For example:**




Here in the above diagram, for every member of $X$ there exists a unique matching member $Y$. Thus, the above function is one-to-one.

Now, consider the first table given in the textbook draw the diagram for showing the matching between the members.

Consider the following diagram:




Here, in the above diagram for every member of $X$ there exists a matching member $Y = f(X)$.

But in this case:

$$f(1) = f(3) = f(5) = 6$$
$$f(2) = f(4) = 7$$

So the function $f$ is not a one-to-one function.

Comment

**Step 2** of 7

b. **Onto function:**

A function $f : X \to Y$ is said to be onto if and only if when for each and every member of $y \in Y$ there exist a matching member $x \in X$ with the property that $y = f(x)$ where $x \in X, y \in Y$.

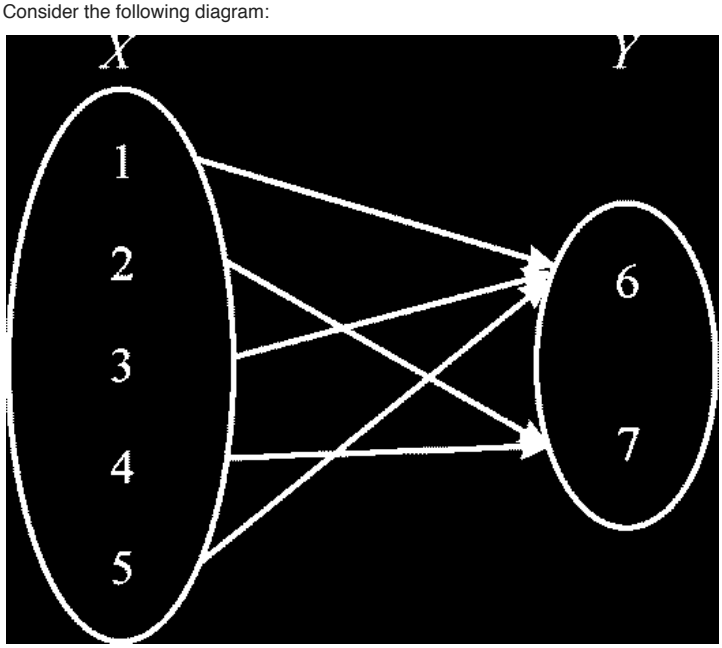
In other words it can be described as follows:

The size of f(x) should be equal to size of the set Y.

It is also known as surjective function.

Now, consider the first table given in the textbook draw the diagram for showing the matching between the members.

Consider the following diagram:



Here, in the above for each and every member of $y \in Y$ there does not exist a matching member $x \in X$ with the property that $y = f(x)$ where $x \in X, y \in Y$.

$$f(1) = f(3) = f(5) = 6$$
$$f(2) = f(4) = 7$$

and the elements 8, 9 and 10 in set Y are left out without any elements in $x \in X$ such that $y = f(x)$

So the function $f$ is not an onto function.

Comments (6)

**Step 3** of 7

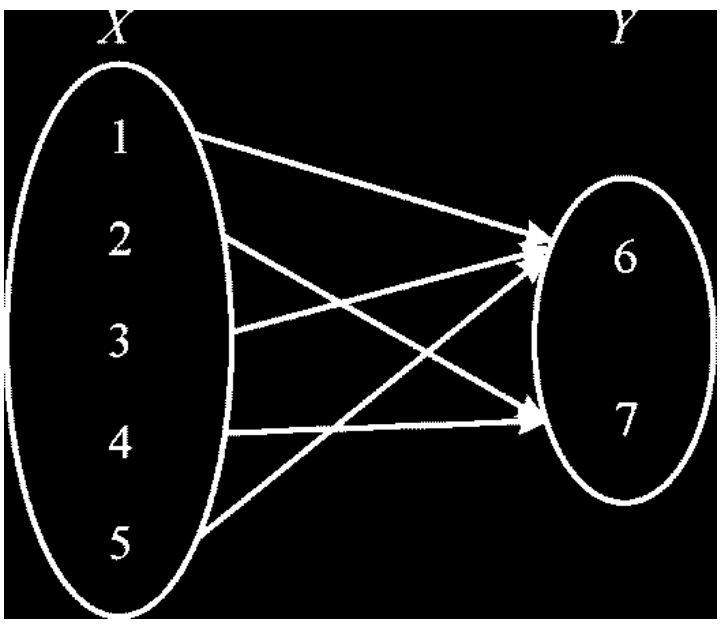
c. **Correspondence:** A function $f : X \to Y$ is said to be one to one correspondences when this function is one to one and onto at the same time. It is also known as bijective function.

Now, consider the first table given in the textbook draw the diagram for showing the correspondence between them.

Consider the following diagram:

Comment

**Step 4** of 7




It is clear from the above diagram that the function $f : X \to Y$ is not a one-to-one and not onto. So the function $f : X \to Y$ is not one-to-one and not onto function.

So, the function $f$ is not a correspondence.

Comment

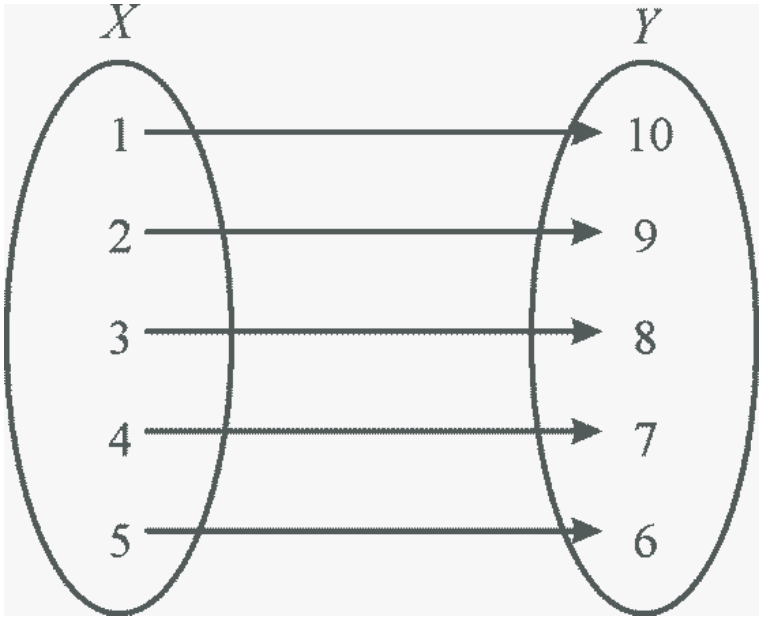
**Step 5** of 7

d. **One-to-one Function:** A function $g : X \to Y$ is said to be one-to-one if and only if when for each and every member of $X$ there exist a unique matching member $Y$.

It is also known as injective function.

Now, consider the first table given in the textbook draw the diagram for showing the matching between the members.

Consider the following diagram:




Here in the above diagram, for every member of $X$ there exists a unique matching member $Y = f(X)$.

So, the function $g$ is a one-to-one function.

Comment

**Step 6** of 7

e. **Onto function:**

A function $f : X \to Y$ is said to be onto if and only if when for each and every member of $y \in Y$ there exist a matching member $x \in X$ with the property that $y = f(x)$ where $x \in X, y \in Y$.

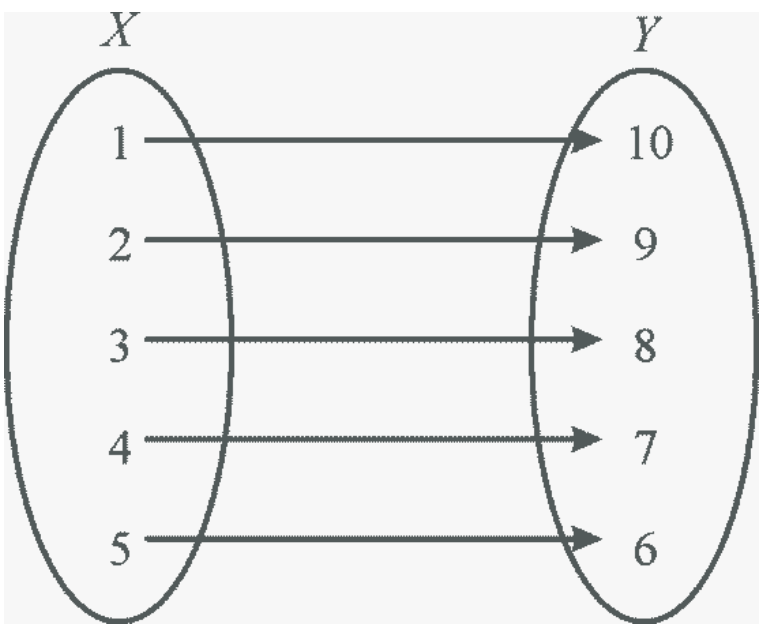
In other words it can be described as follows:

The size of f(x) should be equal to size of the set Y.

It is also known as surjective function.

Now, consider the first table given in the textbook draw the diagram for showing the matching between the members.

Consider the following diagram:




Here in the above diagram, for every member of $X$, there exists a matching member $Y$ with the property $y = f(x)$.

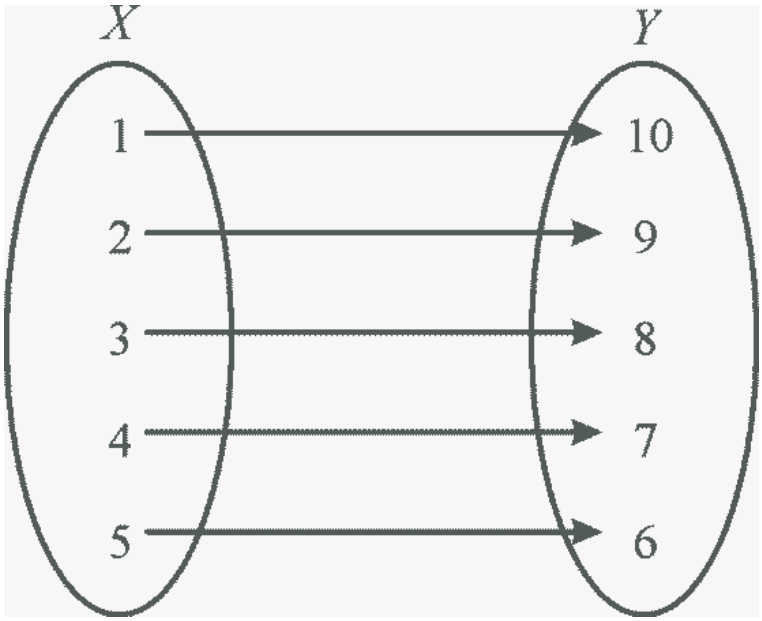
So the function $g$ is a onto function

Comment

**Correspondence:** A function $g: X \to Y$ is said to be one to one correspondences when this function is one to one and onto at the same time. It is also known as bijective function.

Now, consider the first table given in the textbook draw the diagram for showing the correspondence between them.

Consider the following diagram:




It is clear from the above diagram that the function $g: X \to Y$ is one to one and onto at the same time.

So the function $g$ is a correspondence.

Comment

## Problem

Next

Let β be the set of all infinite sequences over {0,1}. Show that β is uncountable using a proof by diagonalization.

## Step-by-step solution

**Step 1** of 3

Let $\mathcal{B}$ be the set of all infinite sequences over {0,1}. Every element in $\mathcal{B}$ is an infinite sequence $(a_1, a_2, a_3, \ldots)$ where $a_i \in \{0,1\}$. Assume $\mathcal{B}$ is countable. Define a correspondence *f* between $\mathcal{B}$ and $N = \{1,2,3,\ldots\}$.

Comment

**Step 2** of 3

Suppose, for $z \in N$, $f(z) = (a_{z1}, a_{z2}, a_{z3}, \ldots)$ where $a_{zi}$ is defined as the $i^{th}$ bit in the $z^{th}$ sequence.

In other words,

| $z$ | $f(z)$ |
|---|---|
| 1 | $(a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, \ldots)$ |
| 2 | $(a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, \ldots)$ |
| 3 | $(a_{31}, a_{32}, a_{33}, a_{34}, a_{35}, \ldots)$ |
| 4 | $(a_{41}, a_{42}, a_{43}, a_{41}, a_{45}, \ldots)$ |
| $\vdots$ | $\vdots$ |

Comment

**Step 3** of 3

Now, a sequence $b$ is defined in such a way that $b = (b_1, b_2, b_3, \ldots)$ belongs to $\mathcal{B}$ over $\{0,1\}$ where $b_i = 1 - a_{ii}$ for $i \in N$.

Consider the following example,

| $z$ | $f(z)$ |
|---|---|
| 1 | $(0,1,1,0,0,\ldots)$ |
| 2 | $(1,0,1,0,1,\ldots)$ |
| 3 | $(1,1,1,1,1,\ldots)$ |
| 4 | $(1,0,0,1,0,\ldots)$ |
| $\vdots$ | $\vdots$ |

The sequence *b* is computed as $b = (1-a_{11}, 1-a_{22}, 1-a_{33}, 1-a_{44}, \ldots) = (1,1,0,0,\ldots)$. Therefore, $b \in \mathcal{B}$ is different from every sequence by minimum one bit. Thus, $b$ is not equal to $f(z)$ for any $z$. It is a contradiction that $\mathcal{B}$ is uncountable.

**Therefore, $\mathcal{B}$ is uncountable.**

Comment

## Problem

Let $T = \{(i, j, k) | \, i, j, k \in \mathcal{N}\}.$ Show that T is countable.

## Step-by-step solution

**Step 1** of 5

**Countability:** A set is countable if a set is either finite or has the same cardinality as the set of positive integers.

Comment

**Step 2** of 5

Given $\mathrm{T} = \{\langle i, j, k \rangle | \mathrm{i, j, k} \in \mathrm{N}\}$

• The goal is to prove that $T$ is a countable set.

• First, let's define a set P = $\{\langle i, j, k \rangle \in T \,|\mathrm{i+j+k = s}\}$ for each triple $\langle \mathrm{i, j, k} \rangle$ where $\mathrm{i, j, k} \in \mathrm{N}$, let $\mathrm{i+j+k}$ be the sum s of the triplet.

• Now, for each number $\mathrm{s} \in \mathrm{N}$,

• There are finitely many triples that has sum equal to s.

• Enumerating the triples with sum zero, then triples with sum equal to 1, then sum equal to 2 and so on.

• The previous step will follow all the triples in T.

• Hence, set P is finite for every $\mathrm{s} \in \mathrm{N}$.

Comment

**Step 3** of 5

Now, since P is finite and according to the given definition, it is countable too, therefore the set $P' = \{\langle i', j', k' \rangle \in T \,|\mathrm{i'+j'+k' = s'}\}$ is also countable.

Comment

**Step 4** of 5

Therefore, any set $P_i$ where $i \in N$, the union $T = \bigcup_{i \in N} P_i$ is also countable since, a countable union of a number of finite sets is countable.

Comment

**Step 5** of 5

**Hence, it is proved that T is countable.**

Comment

## Problem

Review the way that we define sets to be the same size in Definition 4.12 (page 203). Show that "is the same size" is an equivalence relation.

**DEFINITION 4.12**

Assume that we have sets $A$ and $B$ and a function $f$ from $A$ to $B$. Say that $f$ is ***one-to-one*** if it never maps two different elements to the same place—that is, if $f(a) \neq f(b)$ whenever $a \neq b$. Say that $f$ is ***onto*** if it hits every element of $B$—that is, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. Say that $A$ and $B$ are the ***same size*** if there is a one-to-one, onto function $f: A \longrightarrow B$. A function that is both one-to-one and onto is called a ***correspondence***. In a correspondence, every element of $A$ maps to a unique element of $B$ and each element of $B$ has a unique element of $A$ mapping to it. A correspondence is simply a way of pairing the elements of $A$ with the elements of $B$.

## Step-by-step solution

**Step 1** of 3

**Given:**

A function $f: A \to B$ in which $A$ and $B$ are two sets. The function $f: A \to B$ is **one to one function** as it never maps two different elements of same set with one element of another set.

The function $f: A \to B$ is **onto function** as each and every element of the set $B$ is hit by the element of set $A$.

As the function $f: A \to B$ is one to one and onto at the same time then it means the set $A$ and $B$ has the same cardinality. If the cardinality of these two sets is same so these sets are of **same size or equinumerous**.

If the function $f: A \to B$ is one to one and onto at the same time it means the function $f: A \to B$ is **correspondence function** also. **Correspondence function** is also known as **bijective function**.

If the function $f: A \to B$ is one to one and onto or bijective function, then sets $A$ and $B$ are of **same size**.

Comment

**Step 2** of 3

**Proof:**

**Equivalence relation:** A relation is known as equivalence in nature if it is reflexive, transitive and symmetric.

**Same size** relation is equivalence relation if and only if it is symmetric, reflexive and transitive.

• **For reflexivity:** If the user checks the identity function on the set $A$ then this identity function is a bijection from $A$ to $A$.

Hence the **same size** relation is reflexive relation.

• **For symmetry: if the function** $f: A \to B$ is a bijective function then it means the inverse function $f^{-1}$ is also bijective function from the set $B$ to set $A$.

Hence if $A \sim B$ then $B \sim A$, so **the same size** relation is symmetric relation also.

• **For transitivty: Assume that** $A \sim B$ and $B \sim C$. Then the function $f: A \to B$ is bijective function from A to B and the function $g: B \to C$ from $B$ to $C$.

Therefore the composition of two bijective functions $f$ and $g$ is also a bijective function from $A$ to $C$.

Hence the **same size** relation is transitive relation as $A \sim C$.

Comment

**Step 3** of 3

**Conclusion:**

Hence the **same size** relation is reflexive, symmetric and transitive in nature so the **same size** relation is equivalence relation.

Comment

## Problem

Let $INFINITE_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) \text{ is an infinite language}\}$. Show that $INFINITE_{DFA}$ is decidable.

## Step-by-step solution

**Step 1** of 1

**Decidability of Infinite $_{DFA}$ can be proved as follows:**

For deciding $INFINITE_{DFA}$ construct a Turing machine $T1$.

*T1= "On input* $\langle A \rangle$ *, where A is* Deterministic Finite Automata.

1. Assume the value of *n* as the number of states of A.

2. Build Deterministic Finite Automata *M1* that accepts all the strings of length *n* or more.

3. Build a Deterministic Finite Automata *M2* such that *L (M2) = L (A)* $\cap$ *L (M1)*

4. Test *L (M2) =* $\phi$ using the $E_{DFA}$ decider *T* from the theorem 4.4.

5. If *T* accepts reject; if *T* rejects, accept."

The above algorithm functions because a Deterministic Finite Automata that accepts infinitely many strings must accept randomly long strings.

Hence, the algorithm recognizes such Deterministic Finite Automata.

Contrariwise, if the algorithm recognizes a Deterministic Finite Automata, the Deterministic Finite Automata recognizes certain strings of length *n* or more where *n* is the number of states of the DFA.

This string may be pumped in the method of the pumping lemma in favor of the regular languages to find several recognized strings.

Since Turing machine $T1$ decides $INFINITE_{DFA}$ so $INFINITE_{DFA}$ is decidable.

Comment

## Problem

Let $INFINITE_{PDA}$

$$= \{\langle M\rangle \mid M \text{ is a PDA and } L(M) \text{ is an infinite language}\}.$$ Show that $INFINITE_{PDA}$ is decidable.

## Step-by-step solution

**Step 1** of 1

Given that

$$Infinite_{PDA} = \{ \langle M\rangle \mid \text{M is a PDA and L(M) is an infinite language } \}$$

**The decidability of *Infinite* $_{PDA}$ is as follows:**

• To decide the $\text{Infinite}_{PDA}$ convert the given PDA *M* into a CFG *G*.

• Now convert into an equivalent grammar $G'$ in Chomsky Normal Form (CNF).

• Generate a graph A1: Each Variable is a vertex, and for a rule or production

• $T \to UV$ add edges (T, U) from T to U. and (T, V) from T to V.

• Apply DFS or BFS from the start state to see if A1 has a directed cycle. If it does, accept. Otherwise reject.

• If $\langle M\rangle \in \text{INFINITE}_{\text{PDA}}$, then it has a string of length greater than pumping length of L (M).

• Following the proof of pumping lemma, it means there is a Variable V which can derive sVt for some strings s, t. (Since it is in CNF it must be non-empty).

• This implies that the graph A1 has a cycle involving the variable V.

• Assume there is a cycle in A1. Then it is clear that for some variable V, a derivation $V \to {}^*sVt$ must exist and s or t must be a non-empty, since $G'$ is in CNF.

• Since there is a scope for finding a cycle from S. there is rule $S \to {}^*aVb.$

• Thus, $S \to {}^*au^iVv^ib.$ for all i>=0 and so L(M) is infinite.

Hence, basing on the above discussion it is proved that $Infinite_{PDA}$ is decidable.

Comment

## Problem

Let A = { $\langle M \rangle |\ M$ is a DFA that doesn't accept any string containing an odd number of 1s}. Show that A is decidable.

## Step-by-step solution

**Step 1** of 2

Consider a language,

$A = \{< M >| M \text{ is a DFA which does not accept any string containing an odd number of 1s}\}$

The language is said to be decidable if there exists a Turing machine for it. Construct a Turing machine for A to check the decidability.

Comment

**Step 2** of 2

The Turing machine for A is as follows:

$I =$ "On input $<M>$ where $M$ is a DFA:

1. Construct a new DFA $D_X$ that accepts any string containing an odd number of 1s.
2. Construct another DFA $D_Y$ such that $L(D_Y) = L(M) \cap L(D_X)$.
3. Check whether $L(D_Y) = \phi$, using the $E_{DFA}$ decider $T$.
4. If $T$ accepts, accept; otherwise reject."

**There exists a Turing machine for A. Therefore, A is decidable.**

Comment

## Problem

Let A = { $\langle R, S \rangle |$ } Show that A is decidable.

## Step-by-step solution

**Step 1** of 3

The idea of the proof is as follows:

• L(R) is a subset of L(S) iff L(R) intersected with the complement of L(S), $L(S)^C$ is the empty set.

• Use the Theorem 4.4 (or its equivalent for regular expressions) to prove that

the condition "L(R) intersection $L(S)^C$ = empty set" is decidable.

Comment

**Step 2** of 3

**Proof:**

The following Turing machine T decides the language A.

"On input string *w*,

1. Check that *w* encodes a pair $< R, S >$ where R and S are regular expressions

If it does not, then reject *w*.

2. Translate R into an equivalent DFA DR and S into an equivalent DFA DS.

3. Build a DFA $DS^C$ that accepts the language $L(DS)^C$.

4. Build a DFA D that is the intersection of DR with $DS^C$ that is, L(D) = L(DR) intersection $L(DS^C)$.

5. Now, run the Turing machine T with input < D > to determine if L(D) is empty.

If T accepts < D > (which means that L (D) is empty), then accept *w*.

If T rejects < D > (which means that L (D) is NOT empty), then reject *w*."

Comment

**Step 3** of 3

Now prove that the Turing machine T decides the language A.

• First of all, prove that T halts on all inputs.

• Let *w* be any word. There are two possibilities.

• Either *w* codifies a pair of regular expressions or it doesn't. Checking this will take a finite amount of time.

• If x doesn't codify a pair of regular expressions, then T stops rejecting *w*.

• Otherwise, assume *w* codifies a pair < R, S >.

• Constructing the DFAs described in steps 2, 3, and 4 above is possible (regular languages are closed under intersection and complementation) and takes finite time.

Hence, in all cases T stops on any input *w* after a finite amount of time.

Hence, it is proved that language A is decidable.

Comments (2)

## Problem

Let ∑ = {0,1}. Show that the problem of determining whether a CFG generates some string in $1^*$ is decidable. In other words, show that

$$\{\langle G\rangle \mid G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

is a decidable language.

## Step-by-step solution

**Step 1** of 3

• Given a CFG over $(0,1)$ that generates **some** string in $1^*$. The language generated by this CFG, denoted by Language A can be rephrased in the following way: $A=\{<G>|G \text{ is a CFG over } (0,1)^* \text{ and } 1^* \cap L(G) \neq \varnothing\}$.

• **To test decidability, the fact that intersection of a Regular Language (RL) & Context Free Language (CFL) is a CFL shall be harnessed.**

Comment

**Step 2** of 3

• Note that $1^*$ is a Regular Language & $L(G)$ is a CFL (since G is a CFG), therefore, $1^* \cap L(G)$ is clearly a CFL.

• A Turing Machine (TM) has the task to test whether the problem at hand is decidable or undecidable.

• If for an input, TM culminates in either ACCEPT or REJECT state, the problem is referred to as **decidable**.

Comment

**Step 3** of 3

Let H be the TM that decides A. Using the Theorem 4.8, the decidability is employed by the following algorithm:

H="on input <G>, where G is a CFG":

Construct a CFG B, such that $L(B) = 1^* \cap L(G)$ (remember that $1^* \cap L(G)$ is CFL so the statement is valid).

1. Run the TM R that decides $E_{CFG}$ on $< B >$. It may be elaborated here that $E_{CFG}$ denotes the problem of determining whether a CFG (here B) generates any strings at all is decidable. Formally,

$$E_{CFG} = \{< B >| B \text{ is a CFG and } L(B) = \varnothing\}$$

2. **If R accepts, reject**. It means that the language generated by the CFG B is empty. Therefore, TM H culminates in REJECT state.

3. **If R rejects, accept**. In other words, the problem at hand is Decidable since language generated by the CFG *B* is NOT empty.

**Elaboration**: If TM R accepts, it would imply that,

$$\begin{aligned} L(B) &= 1^* \cap L(G) \\ &= \varnothing \end{aligned}$$

That, is for some string $w \in 1^*$, $w \notin L(G)$, hence R should reject.

Comment

## Problem

Show that the problem of determining whether a CFG generates all strings in $1^*$ is decidable. In other words, show that

$$\{\langle G\rangle \mid G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \subseteq L(G)\}$$

is a decidable language.

## Step-by-step solution

**Step 1** of 1

**Given:**

$S=\{\langle G\rangle \mid G\}$ is a context free grammar over the string $\{0,1\}$ and $1^* \subseteq L(G)$

User need to proof that string $1^*$ is generated by CFG and it is decidable.

**Proof:**

First the assumption is made that grammar *G* is in the CNF. If it is not present in the CNF then Turing machine should be use to make the grammar in CNF.

Suppose there is total *n* variable in the grammar *G* and the grammar there is one pumping lemma constant in grammar. Assume pumping lemma constant is *s* and the value of this pumping lemma constant is assumed to be $2^{n-1}$.

It is prove that if the value of *k* is greater than or equal to *s*. If the grammar *G* can generate $1^k$ then it would be capable of generating $1^{k+s}$.

Grammar *G* is capable of generating string which is in the form $1^m$ for $0 \le m < s!+s$ then G is use for generating $\Sigma^*$.

Turing machine is use for verifying that whether the string is accepted or not. If the string is recognized then Turing machine accepts that particular string otherwise Turing machine reject that particular string.

**Construction:**

If the Grammar *G* is use for generating $1^k$ then the grammar *G* should also generates $1^{k+p!}$

With the help of the lemma $1^k$ is divided into the multiple string named *a*, *b*, *c*, *d* and *e* in such a manner that *abcde*=$1^k$. $|bcd| \le s, |bd| > 0$, $ab^icd^ie \in L(G)$ for all the value of *i* which is greater than or equal to zero.

Suppose, $h=|bd|$ in such a way that $1 \le h \le s$.

With the help of lemma $1^{k+ih} \in L(G)$ for the integer *i* greater than or equal to zero.

Hence, $1^{k+ih} \in L(G)$ it implies grammar G generates $1^{k+ih}$.

**Conclusion:**

Turing Machine is working as a decider for CFG and deciding whether the string is accepted or not by using the predefined rules mentioned above.

Hence, it can be said that the Turing machine *S* is decidable.

Comment

## Problem

Let A = { $\langle R \rangle \mid R$ is a regular expression describing a language containing at least one string w that has 111 as a substring (i.e., w = x111y for some x and y)}. Show that A is decidable.

## Step-by-step solution

**Step 1** of 3

Consider the language,

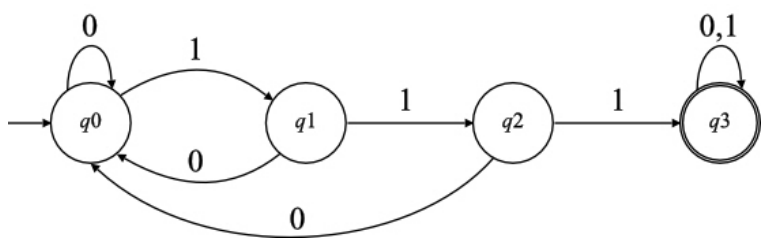
$$A = \left\{ \begin{array}{l} \langle R \rangle \mid R \text{ is a regular expression describing a language which contain} \\ \text{at least one string } w \text{ containing 111 as its substring} \end{array} \right\}$$

**The decidability of the language *A* is proved as follows:**

• Define a language *S* such that $S = \{ w \in \Sigma^* \mid w \text{ consists 111 as a substring} \}$.

• The regular expression (RE) for the language *S* is $(0 \cup 1)^* 111 (0 \cup 1)^*$.

• The DFA $D_S$ for the language *S* is shown below:




Comment

**Step 2** of 3

• Now think about some RE *R* on input alphabet $\Sigma$.

• If $S \cap L(R) \neq \phi$, then *R* produces a string containing $111$ as a substring. Thus, $\langle R \rangle \in A$.

• Similarly, if $S \cap L(R) \neq \phi$ then R produces a string that does not contain 111. Thus, $\langle R \rangle$ does not belongs to *A*.

• Since $L(R)$ is described by regular language, $L(R)$ is a regular language. Both *S* and $L(R)$ are regular languages.

• $S \cap L(R)$ is regular because, regular languages are closed under intersection. Thus, $S \cap L(R)$ has some DFA $D_{S \cap L(R)}$.

• Theorem 4.4 shows that $E_{DFA} = \{ \langle K \rangle \mid K \text{ is a DFA with } L(K) \neq \phi \}$ is decidable. Thus, there exists a Turing Machine *TM* which determines $E_{DFA}$.

• Relate TM *T* to $D_{S \cap L(R)}$ to determine if $L(R) \cap S \neq \phi$.

Comments (2)

**Step 3** of 3

Summarization of the above discussion contributes the subsequent Turing machine *M* to decide *A*:

*M* = "On input $\langle R \rangle$, where R is a regular expression:

• Transform *R* into a DFA $D_R$ by means of the algorithm in the proof of Kleene's Theorem.

• Build a DFA $D_{S \cap L(R)}$ for the language $S \cap L(R)$ from the DFAs $D_S$ and $D_R$.

• Run TM *T* that decides $E_{DFA}$ on input $\langle D_{S \cap L(R)} \rangle$.

- If $T$ accepts, reject. If $T$ rejects, accept.

**The Turing machine $T$ decides $A$ . Therefore, the language $A$ is decidable.**

Comment

## Problem

Prove that $EQ_{DFA}$ is decidable by testing the two DFAs on all strings up to a certain size. Calculate a size that works.

## Step-by-step solution

**Step 1** of 1

**Given:**

Assume that $EQ_{DFA} = \{\langle A,B\rangle | A,B \text{ are } DFAs \text{ and } L(A) = L(B)\}$ and suppose M is a Turing machine for determining the decidability of $EQ_{DFA}$.

**Construction of Turing machine $M$ :**

- $M$ = "On input $\langle A,B\rangle$, where $A$ and $B$ are DFAs with the same terminal symbols. If symbols are not same then reject.
- Calculate the number of states in the DFA, $A$ and $B$ and stored them in $n$ and $m$ respectively.
- Iterate all strings which comes under $\Sigma$ till $n.m$
- Now for each and every string $w$

o Simulate DFA $A$ on the string $w$

o Simulate DFA $B$ on the string $w$

o Here Turing Machine $M$ is working as a decider by running on the output provided both DFA $A$ and $B$. If result of both the simulation comes different the reject the string, otherwise accept it.

- After that all $n.m$ strings then accept it otherwise reject it.
- If $M$ accepts, accept. If $M$ rejects, reject."

**Size for working:**

The reason behind checking first $n.m$ strings is that if both DFAs do not accept the same language so there is a string $w$ of size $|w|$ and it is less than equals to size of $n.m$ for $A(w) \neq B(w)$.

By using contradiction, suppose that the first string provides a different output of DFAs A and B is $w'$. The length $l$ of $|w'|$ is greater than $n.m$.

Now the sequence of states for the DFA $A$ is $a_0, a_1, a_2 \ldots\ldots a_l$ and the sequence of states for the DFA $B$ is $b_0, b_1, b_2 \ldots\ldots b_l$ for describing the transitions for $w'$ in DFA $A$ and $B$.

As the value of l is greater than $n.m$, if the user places above sequences side by side so there is some repetition in pairs like $a_i, b_i$ and $a_j, b_j$ such that $a_i = a_j, b_i = b_j, i < j$ also present.

So user can remove all the sequences remaining $a_i, b_i$ and get a smaller string $w''$. As the DFA $A$ and $B$ is same over as $w'$ thus our contradiction becomes false.

As the string $w''$ has length less than equal to $n.m$ such that $A(w'') \neq B(w'')$.

S**o, checking all the strings up to size $n.m$ is enough.**

**Conclusion:**

Here two Turing Machines are used M is used as decider to find decidability of $A$ and $B$ Turing Machine $M$ is used to find Running condition of Turing machines $M$. This way $A$ and $B$ are being decided by $M$, $A$ and $B$ are components of $EQ_{DFA}$ so, $EQ_{DFA}$ is also decidable.

Comments (1)

## Problem

Let C be a language. Prove that C is Turing-recognizable iff a decidable language D exists such that

$$C = \{x \mid \exists y\ (\langle x, y\rangle \in D)\}.$$

## Step-by-step solution

**Step 1** of 1

**Given:** A language $C$, and this language is Turing recognizable if and only if there exist y such that $\langle x, y\rangle \in D$.

**Proof:**

Assume that a language $C$. This language is recognized by a Turing machine $M$. When the input $x$ is passed to the Turing machine this machine simulates machine $M_d$. The decider for the language $D$ on the input string $\langle x, y_i\rangle$, here $y_1, y_2, y_3 \ldots\ldots$ string in lexicographic order.

Assume that the language $C$ is recognizable. Suppose $L(M) = C$ which means that Turing machine $M$ recognize the language $C$. If $L(M) = C$ then for every string $x$ which belongs to the language $C$, an accepting computation of $M$ on the input string $x$ is present.

Suppose the language $D$ on input $\langle x, y\rangle$ verifies whether $y$ encodes an accepting computation of the input string $x$ on machine $M$.

Example of such encoding is $c_0 \# c_1 \# c_2 \ldots\ldots \# c_n$

Here in the above encoding $c_i$'s configurations of the Turing machine $M$ on the input string $x$. $c_0$ Is the initial configuration of the input string $x$ on machine $M$. The configuration $c_n$ is the accepting configuration for input $x$. After each $c_i$, $c_{i+1}$ come.

So it is clearly seen that $D$ is decider.

**Construction:**

Here user supposed to construct Turing machine that will decide the decidability of $C$. Now follow these terms:

- For proving decidability of $C$ one needs a Turing Machine so consider a Turing Machine $T$.
- Construct Turing Machine in a way so that each possible string of $Y$ can be searched or found.
- Test the string whether it is according to the predefined rules $C = \{x \mid \exists y \langle x, y\rangle \in D\}$ in the question.
- If $\langle x, y\rangle \in D$ then $T$ accepts otherwise rejects.

**Conclusion:**

Here $D$ is Recognizable by $C$ and $C$ is recognizable by Turing machine $T$ and $C$ is also decided by Turing Machine $T$ so $\langle x, y\rangle \in C$ as well this way $C$ is Turing recognizable.

Comment

## Problem

Prove that the class of decidable languages is not closed under homomorphism.

## Step-by-step solution

**Step 1** of 1

**Given:**

Consider a decidable language *K* and a homomorphism *m* to show that $m(K)$ is un-decidable.

**Proof:**

- Assume that $K = \left\{ \begin{array}{l} uv \mid u \in \{0,1\}^*, v \in \{a,b\}^*, u = \langle W, w \rangle, \\ \text{and } v \text{ encodes an integer } n \\ \text{using turing machine } W \text{ having input } w \text{ get halts in } n \text{ steps} \end{array} \right\}$
- As K is decidable so simulate Turing machine *W* on input *w* for *n* steps.
- Consider homomorphism $m(0) = 0$, $m(1) = 1$ and $m(a) = m(b) = \varepsilon$
- $m(K) = \text{HALT}$ which is un-decidable.

Hence, Decidable language *K* is not closed under homomorphism *m*.

Comment

## Problem

Let A and B be two disjoint languages. Say that language C ***separates*** A and B if $A \subseteq C$ and $B \subseteq \overline{C}$. Show that any two disjoint co-Turing-recognizable languages are separable by some decidable language.

## Step-by-step solution

**Step 1** of 1

**Given:**

Two disjoint languages are supposed to be *A* and *B*. Decidable Language *C* is chosen so that it is use for separating *A* and *B* if $A \subseteq C and B \subseteq \bar{C}$. Now, according to this concept two disjoint Co-Turing recognizable languages can be separated with the help of decidable language.

**Proof:**

Consider two DFA *M* and *N* for languages *A* and *B*. *O* is a DFA for separator *C*. Here language *A* is assumed to the subset of *C* and language *B* is assumed to be the subset of $\bar{C}$ . In this way *C* is separating *A* and *B* because *C* or $\bar{C}$ is not deciding both *A* and *B* instead *A* is decided by *C* and *B* is decided by $\bar{C}$. It can be understood that *C* and $\bar{C}$ both are dissimilar and deciding A and B individually.

**Construction:**

Concept can be understood in better by using following approach:

Turing Machine *F* = runs on input > *where A and B are two DFA*

*• Construction of DFA O is done for C. DFA O is working as a decider for language A and B.*

*• Run Turing Machine T on input of C. It will find the decidability capacity of O.*

*• If Turing machine accept the string then it is recognized and if Turing machine reject the string then it is not recognized.*

***Conclusion:***

*Now, if T accepts then DFA O for language C is working as a decider for language A otherwise it is working as decider for language B.*

*Now, it is quite easy to understand two disjoint Co-Turing recognizable languages which are separable by the use of some decidable language.*

Comments (8)

## Problem

Let S = { $\langle M \rangle |$ M is a DFA that accepts $w^R$ whenever it accepts w}. Show that S is decidable.

## Step-by-step solution

**Step 1** of 4

**Given:** $S = \{\langle M \rangle | M \text{ is a DFA that accepts } w^R \text{ whenever it accepts } w\}$.

Here, $M$ is a DFA that accepts $w^R$ whenever it accepts $w$ and $M$ is recognizable and decidable on input $w$.

**Note: -** If a DFA accepts $w^R$ whenever it accepts w, then L(M) = L($M^R$), where $M^R$ is the DFA that accepts the reverse of the strings accepted by M.

Comment

**Step 2** of 4

**Proof that $S$ is decidable is as follows:**

Consider the following Turing Machine T = "On Input M, Where M is a DFA".

1) Construct DFA N which accepts the reverse of a string accepted by M.

2) Submit to the Decider for $EQ_{DFA.}$

3) If it accepts, accept.

4) If it rejects, reject.

T is a Decider since, steps 1, 3 and 4 will not create an infinite loop and step-2 calls a decider. Also, T accept M which is a DFA if L(M) = L($M^R$).

**Therefore, T decides $S$. Thus, $S$ is decidable.**

Comment

**Step 3** of 4

**Construction:**

DFA $M^R$ can be constructed by first constructing NFA from M by reversing all transition in the following way:

• Change the initial state with new accepting state.

• After that, create a new initial start state with $\in$ transition to all earlier accepting state.

• Then construct a DFA from this NFA.

As, a DFA can accept only those particular languages that they are designed for, T is deciding the decidability of M. Decidability or Undecidability of a string depends upon recognition of its components.

It is eventually necessary that output of M is decidable by $S$. So, from the above proof it is clear that the language $S$ is decidable.

Comment

**Step 4** of 4

**Conclusion:**

It is eventually necessary that output of M is decidable by $S$. So, from the above proof it is clear that the language $S$ is decidable.

Comment

## Problem

Let *PREFIX-FREE*$_{REX}$ = { $\langle R \rangle$ | R is a regular expression and L(R) is prefix-free}. Show that *PREFIX-FREE*$_{REX}$ is decidable. Why does a similar approach fail to show that *PREFIX-FREE*$_{CFG}$ is decidable?

## Step-by-step solution

**Step 1** of 3

The prefix free of regular expression is defined as,

$$PREFIX-FREE_{REX} = \{\langle R \rangle \mid R \text{ is a regular expression and } L(R) \text{ is prefix free}\}.$$

The string *x* is said to be a prefix of a string *y* if there exists a string *z* such that *xz*=*y*. If $x \neq y$ then *x* is said to be the proper prefix of *y*. A language is said to be prefix free if none of the strings have proper prefix of its members (i.e., strings).

Comment

**Step 2** of 3

If there exists a TM for any language, then it said to be decidable. Construct the Turing machine *M* to check the decidability of the language.

$M =$ "On input $R$ where $R$ is a regular expression

1. If $R$ is not a valid regular expression then reject.
2. Construct a DFA (Deterministic Finite Automata) for the language accepted by $R$ i.e., L($R$).
3. Run the DFS (Depth First Search) from the start state $q_0$ and remove all the unreachable states from it.
4. For each final state $q_f$, run the DFS to check if there exists path to any other final state from it or any self loop to itself.
5. If there exists a path to another final state or self loop from $q_f$ then reject. Otherwise, accept."

Thus, there exists a TM for $PREFIX-FREE_{REX}$. Therefore, it is decidable.

Comment

**Step 3** of 3

The DFA accepts the prefix free language if and only if there is no accepting string goes through more than one final state. It can be checked by simple graph traversal technique like DFS. In this case, the DFA cannot have two final states for an accepting run. It is decidable.

In the case of context free grammar (CFG), it fails because every context free language cannot have deterministic PDA. In case of regular expressions, every regular expression has a DFA. If the PDA cannot be drawn deterministically for a language, then it cannot be restricted to have a single final state. Therefore, the similar approach fails to prove $PREFIX-FREE_{CFG}$ is decidable.

Comment

## Problem

Say that an NFA is ***ambiguous*** if it accepts some string along two different computation branches. Let $AMBIG_{NFA}$ = { $\langle N \rangle$ | N is an ambiguous NFA}. Show that $AMBIG_{NFA}$ is decidable. (Suggestion: One elegant way to solve this problem is to construct a suitable DFA and then run $E_{DFA}$ on it.)

## Step-by-step solution

**Step 1** of 2

$AMBIG_{NFA}$ **= { is ambiguous NFA} is decidable**

An NFA is ambiguous if it accept string and two or more different computational branches. Decidability of NFA's Ambiguity can be proved by using following approach.

Now, for a given NFA create a DFA D by using strategy NFA-to-DFA conversion which simulates N and accept a string only if it is accepted by NFA and computed using 2 or more computational branches. Now create decider for DFA E for determining whether D accepts any string accepted by NFA N.

Comment

**Step 2** of 2

Simulate N by keeping the pebble on active state. Initially, put red pebble on initial state along $\in$ transaction. As per N's transaction, operations like moving, adding and removing are performed, preserving color of pebbles. Whenever 2 or more pebbles moved to same state replace its pebble with blue pebble. After that input is examined and accepted when either the state of blue pebble is accept state or different accept states have red pebbles.

For state of N, the pebbles have corresponding positions of DFA D. There will be three stages in which position contain blue pebble, red pebble or no pebble. On the basis of these stages, ambiguity is generated then it is decidable by recognizing multiple pebbles for same string. If there are multiple pebbles then it is ambiguous and decidable by multiple pebbles and more than 1 red pebbles.

Comment

## Problem

A ***useless state*** in a pushdown automaton is never entered on any input string. Consider the problem of determining whether a pushdown automaton has any useless states. Formulate this problem as a language and show that it is decidable.

## Step-by-step solution

**Step 1** of 4

Useless states in automata are the **states whose removal from the automata does not make any difference or put any impact on the language accepted by the automaton**.

**Given:** A pushdown automaton.

Comment

**Step 2** of 4

Here, it is required to test whether the machine has useless states or not and whether the problem of testing is decidable or not.

Comment

**Step 3** of 4

• Let *P* be the set of all the strings accepted by a pushdown automaton.

• Let the language $L = \{x \in P \mid x \text{ contains a useless state}\}$.

• To show that language *L* is decidable, construct a Turing machine which accepts strings in language *L*.

• In reference to this book, consider that the question of whether a PDA has an empty language is decidable.

• It can reduce the question of whether a given state *n* is useless to this question by making *n* the only accept state and then determine whether the resulting push down automata has an empty language.

• If it does then, *n* is a useless state.

Comment

**Step 4** of 4

**Hence, our Turing machine successfully decides whether there is any useless state, by performing this test for each and every state in order.**

Comments (2)

## Problem

Let $BAL_{DFA}$ = { ‹M›| M is a DFA that accepts some string containing an equal number of 0s and 1s}. Show that $BAL_{DFA}$ is decidable. (Hint: Theorems about CFLs are helpful here.)

## Step-by-step solution

**Step 1** of 1

**Given:**

$M$ Is a DFA, accepts a string which contains equal number of 0s and 1s.

**Proof:**

For proving $BAL_{DFA}$ is decidable use the concept of context free language which is as follows:

Now, consider a string $w$ that contains equal no of 0s and 1s that is accepted by DFA $M$. If a regular language is Context Free Language then it is decidable and it is decidable then it would be recognizable as well.

Now use the approach in the same way as uses for CFL.

String containing equal no of 0s and 1s can be generated by a particular grammar pattern or can be said by using a Context Free Grammar then it'd be a Context Free Language.

Now, it is already known that each and every context free language is decidable and it can be proved as follows:

Here $M$ is a DFA that accepts string $w$ that contains equal no of 0s and 1s consider $A$ be a CFG that generates $w$.

**Construction:**

Consider a Turing machine that decides $w$, and built a copy of $w$ in Turing machine $A$. This process works as follows:

$T$ = Runs on input $w$

• Run Turing Machine $S$ on input $\langle M, w \rangle$

Here, consider another Turing machine that generates output by considering input from $M$ by running it on input $w$

• If machine accepts, accept; if rejects, reject

**Conclusion:**

The above Turing machine decides $w$ running on $M$, this way $BAL_{DFA}$ is decidable.

Comment

## Problem

Let

$$PAL_{\text{DFA}} = \{\langle M\rangle \mid M \text{ is a DFA that accepts some palindrome}\}.$$

Show that $PAL_{\text{DFA}}$ is decidable. (Hint: Theorems about CFLs are helpful here.)

## Step-by-step solution

**Step 1** of 4

Assume that $PAL_{DFA} = \{\langle M\rangle \mid M \text{ is a } DFA \text{ that accepts some palindrome}\}$. If a Turing machine can be presented for the given DFA that runs finitely and halts, then the $PAL_{\text{DFA}}$ is decidable.

Comment

**Step 2** of 4

Construct a decider *D* for $PAL_{DFA}$ and a Turing machine K that can decides $E_{CFG}$:

D = "On input $\langle M\rangle$,

1. A PDA *P* is constructed as: $L(P) = \{w \mid w \text{ is a palindrome}\}$

2. A PDA *P'* is constructed so that $L(P') = L(P) \cap L(M)$

3. Now $P'$ is converted into an equivalent CFG *G.*

4. Check if $L(G)$ is empty using Theorem 4.8 over Turing Machine K.

5. If $L(G)$ is empty then reject, otherwise, accept.

Comment

**Step 3** of 4

For Turing machine K:

• Both steps 1 and 2 can be done in finite steps.

• Step 3 also takes finite steps to convert $P'$ into its equivalent CFG.

• In step 4, the decider *K* checks whether the language $L(G)$ is empty or not. It can also be done in a finite step.

Comment

**Step 4** of 4

**Since D takes finite steps for any input, it means that it is a decider. Hence, $PAL_{DFA}$ is decidable.**

Comment

## Problem

Let

$$E = \{\langle M \rangle \mid M \text{ is a DFA that accepts some string with more 1s than 0s}\}.$$

Show that E is decidable. (Hint: Theorems about CFLs are helpful here.)

## Step-by-step solution

**Step 1** of 1

**Proving the decidability of the language**

**Given:**

$E = \{\langle M \rangle \mid M\}$, is a DFA which is use for accepting a string *w* which contain more number of 1s and 0s.

**Proof:**

In order to prove that *E* is decidable, context free language should be use. As it is well known that if any string contain more number of 1's and 0's then it is context free language.

Now, consider a string *w* that contains more number of 1s than 0s that is accepted by DFA *M*. If a regular language is Context Free Language then it is decidable and it is decidable then it would be recognizable as well.

Now use the approach in the same way as uses for CFL.

String containing more number of 1s than 0s can be generated by a particular grammar pattern or can be said by using a Context Free Grammar then it would be a Context Free Language.

Now, it is already known that each and every context free language is decidable. Now prove that the context free language is decidable.

**Construction:**

*M* is a DFA that accepts string *w* that contains more number of 1s than 0s. Consider *A* be a CFG that generates *w*. Consider a Turing Machine that decides *w*. and built a copy of *w* in Turing Machine *A*.

Working of the Turing machine process is as shown:

*T* = Runs on input *w*

• Run Turing Machine *S* on input $\langle M,w \rangle$. Here, consider another Turing machine that generates output by considering input from *M* by running it on input *w*.

• If the string is accepted, then Turing machine accept that particular string.

• If the string is not accepted, the Turing machine rejects the particular string.

**Conclusion:**

So, by the above construction it is proved that string is decided by the Turing machine. Hence, it is said that the language *E* is decidable.

Comment

## Problem

Let

$$C = \{\langle G, x\rangle \mid G \text{ is a CFG } x \text{ is a substring of some } y \in L(G)\}.$$

Show that C is decidable. (Hint: An elegant solution to this problem uses the decider for $E_{CFG}$.)

## Step-by-step solution

**Step 1** of 3

Suppose $C = \{\langle G, x\rangle \mid G \text{ is a CFG } x \text{ is a substring of some } y \in L(G)\}$. Now consider the following proof which shows that $C$ is decidable.

A Turing machine $M$ is constructed in such a way that decides $C$ as follows:

• A DFA *A* is constructed in such a way that it is used to recognize that language of the regular expression $\Sigma^* \circ \{x\} \circ \Sigma^*$ (every string with *x* as their substring).

• A DFA *F* is constructed which is used for the context free language $L(G) \cap L(A)$.

• Perform simulation on the Turing machine that decides $E_{CFG}$ on $L(F)$. If it accept, reject, otherwise accept.

Comments (1)

**Step 2** of 3

It is already know that a language is also a context free language if it is an intersection of a regular language and a context free language. Therefore, *F* will be CFG.

• Furthermore, $L(A)$ is a language of every strings with *x* as their substring and it is described above that it is also a regular language.

• So**, if *G* produces some string *y* with *x* as its substring, the intersection, $L(F)$, should be non-empty.**

Comment

**Step 3** of 3

Now it can be concluded from the above that ***C* is decidable.**

Comment

## Problem

Let $C_{CFG}$ = $\{\langle G, k\rangle |$ G is a CFG and L(G) contains exactly k strings where k ≥ 0 or k = 1}. Show that $C_{CFG}$ is decidable.

## Step-by-step solution

**Step 1** of 1

**Decidability**

Consider a decider *M* which is used to check whether language of CFG is finite or infinite. Use another decider that is Turing machine *W* which shows that is $C_{CFG}$ decidable.

1. *W* = "on input $\langle G,k\rangle$" where *G* is CFG and *k* is string

2. Check $L(G)$ is infinite using decider *M.*

• If $L(G)$ is infinite and $k=\infty$, it is accepted

• If $L(G)$ is infinite and $k\neq\infty$, it is rejected

• If $L(G)$ is finite and $k=\infty$, it is rejected

• If $L(G)$ is finite and $k\neq\infty$, continue

3. Calculate the pumping length *l* for grammar *G.*

4. Set $count=0$

5. Use *for* loop $i=0 \text{ to } l$

• Use *for* loop to get all strings *S* whose length equal to *i*

• If *S* can be generated by *G* then make an increment in *count.*

6. Check value of *count* is equal to *k* then it is accept, otherwise reject.

**Explanation:**

• The Step 2 checks whether $L(G)$ is infinite or not. After step 2 there is grammar whose language which has finite set. In order to prove $C_{CFG}$ is decidable there is only need to prove that the size of language is *k.*

• To do so use loop to find the all the possible string can be generated by grammar G. The grammar is finite therefore the length of string cannot be more than pumping length *l.*

• Make an increment in variable count if the string can be generated by grammar *G.*

• In the last step check value of *count* is equal to *k.*

• Now, it has finite number of steps therefore it can easily check.

Thus *W* is decider, therefore $C_{CFG}$ is also decidable language.

Comments (2)

## Problem

Let A be a Turing-recognizable language consisting of descriptions of Turing machines, $\{\langle M_1\rangle, \langle M_2\rangle, \ldots\}$, where every $M_i$ is a decider. Prove that some decidable language D is not decided by any decider $M_i$ whose description appears in A.

(Hint: You may find it helpful to consider an enumerator for A.)

## Step-by-step solution

**Step 1** of 2

Consider the Turing-recognizable language *A* which contains the descriptions of all the Turing machines, therefore there must exist an enumerator *E* to enumerate it.

Consider $\langle M_i\rangle$ is the $i^{th}$ output of *E*. Assume $s_1, s_2, s_3 \ldots, s_i$ are the all possible strings of $\{0,1\}^*$. It means $s_1, s_2, s_3 \ldots, s_i$ are made up of combinations of 0's and 1's.

Comment

**Step 2** of 2

Consider a decidable language *D* is defined as follows:

For a string $S_i$,

- If $\langle M_i\rangle$ accepts then $S_i$ does not belongs to the language *D.*
- If $\langle M_i\rangle$ rejects then $S_i$ belongs to the language *D.*

Here, the language *D* is a decidable language and its decider is not present in the list. Therefore, it is proved that there is a decidable language *D* whose decider is not present in *A*.

Comment

## Problem

Next

Say that a variable A in CFL G is ***usable*** if it appears in some derivation of some string w ? G. Given a CFG G and a variable A, consider the problem of testing whether A is usable. Formulate this problem as a language and show that it is decidable.

## Step-by-step solution

**Step 1** of 3

Consider the variable *A* in CFG *G* is usable if it appears in some derivation of some string $w \in L(G)$.

**Proof:**

Let's consider the problem to find whether *A* is usable and formulate this problem as a language and show that it is decidable. The language is:

$USABLE_{CFG} = \{\langle G, A\rangle : G \text{ is a CFG}, A \text{ usable for } G\}$.

Comment

**Step 2** of 3

For CFG *G*, *A* variable is usable if the derivation of: $s \Rightarrow *xAy \text{ of } G$ such that $L(x), L(A) \text{ and } L(y)$ are all nonempty, is exist.

Decidability of language can be proved with the help of some steps. There are two further steps required to find decidability of the language. Now, according to the question:

• Consider, a usable CFG is $A_{CFG}$ that generates decidable language.

• If usable $A_{CFG}$ is decidable then language will be decidable as well.

Comment

**Step 3** of 3

**Construction:**

Now construct and use the approach to find the decidability of $A_{CFG}$ and to find decidability of the language.

Consider a decider *M* for $A_{CFG}$ and $L(x), L(A) \text{ and } L(y)$ to decide *M* and also assume a Turing Machine *R* that will decide, whether output provided by $M_1$ can be accepted or not.

$B = \text{"on input} < M >, \text{where } M \text{ is a CFG"}$, *M* is another CFG that is working as a decider for $A_{CFG}$, Now

• Run Turing Machine *R* on input $<M,M_1>$, where $M_1$ is a CFG that generates $\in^*$, and working as decider for *M1*.

• If R acceptable then accepts, else reject.

**Conclusion:**

Here, it is quite obvious $M_1$ is working as a decider for *M* and *M* is working as a decider for $A_{CFG}$ so $A_{CFG}$ is decidable and then language produced by $A_{CFG}$ will also be accepted by Turing machine so it will also be decidable.

Comment

## Problem

The proof of Lemma 2.41 says that (q, x) is a *looping situation* for a DPDA P if when P is started in state q with x ? $\Gamma$

## Step-by-step solution

**Step 1** of 4

A language $Q$ is said to be **decidable** if a Turing machine (which is also called as decider) $M$ exists in which $Q$ **is accepted** and **halts on every string input.** A language, that is decidable, is also known as a recursive language**.**

• Now, consider a string $l$, then for any string $l$ , there exist the following condition:

- $l \in Q \Rightarrow M$ , **Halts in a state of accepting**.
- $l \notin Q \Rightarrow M$ , **halts in a state of non-accepting.**

It is also known that "each language which is decidable is also **Turing-acceptable**".

Comment

**Step 2** of 4

It is already known that "A looping condition for a $\mathrm{DPDA}$ (Deterministic Push Down Automata) $P$ is $(q,x)$ if state $q$ is the starting point of $P$ with $x \in \mathrm{T}$ which exists on the stack's top and everything under $x$ is never popped. Also, an input symbol is never be read".

• The above given problem of looping can be solved by identifying the looping situation. In other words," only in which no further inserted symbol is ever read and re-programming $\mathrm{DPDA}$. Therefore, instead of looping, it rejects and reads the input.

Comment

**Step 3** of 4

The situation arises above, produces a halt situation. The following algorithm explained it more carefully:

$T$ ="On input $\langle B \rangle$, where $B$ is DFA:

1. Initial state of $B$ is marked.

2. Repeat this till a marked on new state:

3. Mark a position on which a **transition exists** in to it from anyother state which is marked already.

4. If none of the state is marked as accepted state, it will be **accepted**; otherwise, **rejected**"

The above algorithm produces a halt situation for accept and non-accept state.

Comments (3)

**Step 4** of 4

Now consider **a language** $F$**, which is defined as** $F = \{\langle P,q,x \rangle \mid (q,x)$ **is a situation of looping for** $P$ $\}$**.** Therefore, from the above discussion it can be said that **"the above given language** $F$ **is a decidable language".**

Comment

## Problem

Show that $EQ_{CFG}$ is undecidable.

## Step-by-step solution

**Step 1** of 1

9640-5-1E AID: 1112 I 27/06/2014

RID: 6385 I 07/08/2015

**Undecidable language:**

The problem of determining whether a string or input can be accepted by a Turing machine or not is called Undecidability. The decidability of the Context-free grammar depends on the decidability of the Turing machine.

**Proof to show that $\mathrm{EQ}_{\mathrm{CFG}}$ is undecidable:**

**Step-1:**

Consider a context-free grammar CFG $G_0 = (V, \Sigma, R, S)$ where $V = \{S\}$ and *S* is a starting variable. Assume that there is a rule $S \to lS$ in *R* for every terminal $l \in \Sigma$. The grammar $G_0$ includes a $\in$ notation by using the rule $S \to \in$.

**Example:**

For the CFG, the rules in $G_0$ are defined as $S \to aS \mid bS \mid \in$ over the alphabet set $\Sigma = \{a, b\}$.So, the grammar CFG $G_0$ satisfies all the alphabets in the alphabet set $\Sigma$.

So, $L(G_0) = \Sigma^*$. Thus, the Turing Machine is decidable.

**Step-2:**

Assume that the CFG is decidable by using the Turing machine R that decides $\mathrm{EQ}_{\mathrm{CFG}}$. Construct another Turing machine S which uses R to decide $ALL_{CFG}$ by using the following procedure:

$S = On\ input\ \langle G_0 \rangle,$

1. Run *R* on the input $\langle G_0, G_1 \rangle$. $G_1$ is a CFG, which generates $\Sigma^*$.
2. Accept the grammar, when *R* accepts.
3. Otherwise reject.

Thus, if the Turing machine *R* decides $\mathrm{EQ}_{\mathrm{CFG}}$, *S* also decides $ALL_{CFG}$ which is impossible. So, $\mathrm{EQ}_{\mathrm{CFG}}$ is also undecidable.

Comment

## Problem

Next

Show that $EQ_{CFG}$ is co-Turing-recognizable.

## Step-by-step solution

**Step 1** of 1

Remember that $EQ_{CFG}$ is co-Turing-recognizable language if and only if its complement $\overline{EQ_{CFG}}$ is a Turing-recognizable language.

Now, $\overline{EQ_{CFG}} = A \cup B$, where

$$A = \{ w \mid w \text{ does not have the form } \langle G1, G2 \rangle \text{ for some CFGs } G1 \text{ and } G1 \},$$

$$B = \{ \langle G1, G2 \rangle \mid \text{G1 and G2 are CFGs and } L(G1) \neq L(G2) \}.$$

• *A* contains strings that defy the syntax for encoding $\langle G1, G2 \rangle$) is simple to accept.

• The set B is realized in the following way:

• Transform the G1, G2 to Chomsky Normal Form (CNF).

• Then begin numbering strings in $\Sigma^*$ lexicographically, where $\Sigma$ is the group of terminals for G1, G2.

• For every string *w* numbered, check if it is produced by G1 and by G2.

• If the 2 Context Free Grammars or both of them cannot produce *w*, then TM goes on to generate the next string in the lexicographic order.

• Else, precisely one of the CFGs produces the string, and the TM accepts.

• Therefore, B is a Turing recognizable language.

• It is already proved that Turing-recognizable languages are closed under union, so $EQ_{CFG}$ is Turing-recognizable language.

• Let the list of strings that are listed in lexicographic order are as follows:

s1, s2, s3 . . . over the input $\Sigma^*$.

Now $EQ_{CFG}$ is realized by the following *TM M*:

*M* = "On input $\langle G1, G2 \rangle$, where G1, G2 are Context Free Grammars:

1. Examine if G1, G2 are valid Context Free Grammars. If atleast 1 does not, accept.

2. Transform G1, G2 into corresponding Context Free Grammars $G'_1$, $G'_2$.both into CNF.

3. Replicate the below step 4 for j = 1, 2, 3 . . .

4. Examine if $G'_1$ and $G'_2$ produce $s_j$ if precisely one of them does, *accept.*"

**Hence it is proved that $EQ_{CFG}$ is co-Turing-recognizable language.**

Comment

## Problem

Find a match in the following instance of the Post Correspondence Problem.

$$\left\{\left[\frac{\mathtt{ab}}{\mathtt{abab}}\right], \left[\frac{\mathtt{b}}{\mathtt{a}}\right], \left[\frac{\mathtt{aba}}{\mathtt{b}}\right], \left[\frac{\mathtt{aa}}{\mathtt{a}}\right]\right\}$$

## Step-by-step solution

**Step 1** of 2

Post Correspondence Problem can be considered as an example of undecidability problem concerning with manipulation of strings to find a match.

A match can be found if the string made or created by combining all the symbols of upper side and string made by combining all the symbols of lower side, both are same.

Comment

**Step 2** of 2

Consider the instance of Post Correspondence Problem with the collection of dominos as follows:

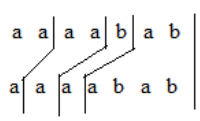
$$\left\{\left[\frac{\mathrm{ab}}{\mathrm{abab}}\right], \left[\frac{\mathrm{b}}{\mathrm{a}}\right], \left[\frac{\mathrm{aba}}{\mathrm{b}}\right], \left[\frac{\mathrm{aa}}{\mathrm{a}}\right]\right\}.$$

The match for the given problem is a sequence of 4, 4, 2, and 1 using the dominos 1, 2 and 4:

$$\left\{\left[\frac{\mathrm{aa}}{\mathrm{a}}\right], \left[\frac{\mathrm{aa}}{\mathrm{a}}\right], \left[\frac{\mathrm{b}}{\mathrm{a}}\right], \left[\frac{\mathrm{ab}}{\mathrm{abab}}\right]\right\}$$

The sequence produces same string aaaabab while reading off the top and bottom of the sequence 4,4,2,1.

The match can be depicted as follows:




**Therefore, a match 4,4,2,1 is found for the given Post Correspondence Problem.**

Comments (2)

## Problem

If A $\leq_m$ B and B is a regular language, does that imply that A is a regular language? Why or why not?

## Step-by-step solution

**Step 1** of 1

No, *A* is not a regular language.

• Assume that the languages *A* is defined as follows:

$A = \{ a^n b^n \mid n \geq 0 \}$ *and* $B = \{b\}$, over the input $\Sigma = \{a, b\}$.

• Specify the function $f : \Sigma^* \to \Sigma^*$ in the following way:

$$f(w) = \begin{cases} b & \text{if } w \in A, \\ a & \text{if } w \notin A. \end{cases}$$

• Notice that if *A* is a context-free language, then it is Turing-decidable.

• Therefore, f is a computable function.

• Besides, $w \in A$ if and only if $f(w) = b$, which is true if and only if $f(w) \in B$.

Hence it is proved that language A is not-regular, but language B is a regular language, because it is finite.

Comment

## Problem

Next

Show that $A_{TM}$ is not mapping reducible to $E_{TM}$. In other words, show that no computable function reduces $A_{TM}$ to $E_{TM}$. (Hint: Use a proof by contradiction, and facts you already know about $A_{TM}$ and $E_{TM}$.)

## Step-by-step solution

**Step 1** of 3

Refer theorem 5.2 in the textbook. It states that $E_{TM}$ is undecidable. It is known that $A_{TM}$ is Turing recognizable but not co-Turing recognizable.

Comment

**Step 2** of 3

The complement of $E_{TM}$ is denoted with $\overline{E_{TM}}$. The TM for $\overline{E_{TM}}$ is as follows:

M="On input <M>,

1. For $l$ = 1, 2, 3, ...
   a. Run M on all strings of length $l$ for $l$ steps.
   b. If any string is accepted then accept.
2. Reject if no string is accepted."

There exists a TM that recognizes $\overline{E_{TM}}$. Thus, $E_{TM}$ is co-Turing recognizable.

Comment

**Step 3** of 3

Assume that $A_{TM}$ is mapping reducible to $E_{TM}$. Thus, $\overline{A_{TM}}$ is mapping reducible to $\overline{E_{TM}}$. $\overline{A_{TM}}$ is not Turing recognizable but $\overline{E_{TM}}$ is Turing recognizable which is a contradiction to the theorem 5.28. This a contradiction to the earlier assumption.

Therefore, $A_{TM}$ is not mapping reducible to $E_{TM}$.

Comment

## Problem

Show that $\leq_m$ is a transitive relation.

## Step-by-step solution

**Step 1** of 2

**Proving relation is transitive**

Transitive relations are those in which one element is related to the second element and second element is related to the third element.

In this situation, the first element should be related to the third element. If the condition is fulfilled, then it can be said that the relation is transitive.

Comment

**Step 2** of 2

Now, for proving transitivity of $\leq_m$ two relations are required.

Consider the first relation for showing the transitivity:

$$A \leq_m B$$

Consider the second relation for showing the transitivity:

$$B \leq_m C$$

Now, computational function for first relation is as shown:

$$x \in A \Leftrightarrow f(x) \in B$$

Here, *x* is considered as input string on *A* and output of *A* is *f(x)*

Now, computational function for second relation is as shown:

$$y \in B \Leftrightarrow g(y) \in C$$

Here, *y* is considered as input string on *B* and output of *B* is *g(y)*

Now, consider a composition function that can be considered as the mapping function between *f* and *g*.

$$h(x) = g(f(x))$$

Now, find the mapping between these two functions, it may be required to create a Turing Machine that computes *h* as follows:

• Simulate a Turing Machine on input *x* and call the output *y*, this will simulate a Turing Machine for *g* on *y*.

• Now, the output of this input can be viewed as considered function that is, $h(x) = g(f(x))$.

In this way:

$$x \in A \Leftrightarrow h(x) \in C$$

Thereby, $h(x)$ is computable function.

Here, input *x* on *A* is directly dependent on *h(x)*.

Hence, if $A \leq_m B$ and $B \leq_m C$ then $\mathrm{A} \leq_{\mathrm{m}} \mathrm{C}$.

Therefore, $\leq_m$ is Transitive Relation.

Comment

## Problem

Show that if A is Turing-recognizable and $A \leq_{\mathrm{m}} \overline{A},$ then A is decidable.

## Step-by-step solution

**Step 1** of 1

**Proving decidability of language**

Consider the Turing machine *M* which is use for recognizing the language *A* in such a way that $A = A(M)$.

So, it can be said that language *A* is Turing-recognizable or even it can be said that it is recognizable.

A Turing machine is use for deciding the language *A* if $A = A(M)$ and Turing machine *M* hold for each and every input.

So, it can be said that *A* is decidable if and only if Turing machine *M* is use for deciding *A*.

Suppose, $A \leq_m \overline{A}$, then it is quite obvious $\overline{A} \leq_m A$ also exists by using the same mapping reducibility function.

*A* is Turing recognizable then $\overline{A}$ is also recognizable as follows:

Assume *M* is the recognizer for Turing Machine $\overline{A}$ and *N* is the recognizer for *A*. *F* is the reduction function for *A* to $\overline{A}$.

*N* can be described as:

*N* is the recognizer and recognizes input or string *w*.

*N* = Input *w*:

• Compute *F(w)*: *F(w)* function is mapping function that computes mapping reducibility between Turing Machines *P* and *Q*.

• Run *M* on input *F(w)* and output whatever *M* output.

As *M* is the recognizer for $\overline{A}$, now run the output *F(w)* on *M* to Find Mapping reducibility between Turing Machines *A* and $\overline{A}$.

This implies that $\overline{A}$ is also Turing Recognizable.

If $A$ and $\overline{A}$ is Turing recognizable then it can be proved that $A \leq_m \overline{A}$ is also decidable.

A language is decidable if its components are Recognizable or co-recognizable as it is already proved that *A* and $\overline{A}$ both are recognizable then consider *P* for deciding the language *A*.

Let $P_A$ and $\overline{P_A}$ is use for deciding that $A$ and $\overline{A}$ is recognizable.

• For any value of input *x* whether it is 1,2, 3 user need to simulate the value for $P_A$ and $\overline{P_A}$ for the finite number of steps. If $x \in A$ then simulation is accepted and if there is the situation that $x \notin A$ then simulation is halted.

• Run both the decider $P_A$ and $\overline{P_A}$ in parallel for the particular input *x* till either of them accepts.

• If $\overline{P_A}$ is accepted then accept it for the particular value of *x* and then halt. If $\overline{P_A}$ is accepted then reject the particular value of *x* and after that halt the Turing machine.

Running $P_A$ and $\overline{P_A}$ in parallel means Turing Machine have 2 tapes 1 for simulating $P_A$ and another for simulating $\overline{P_A}$ it continue until one of them accepts.

Now, it is quite obvious that input *x* is whether running on $P_A$ or $\overline{P_A}$ so it must be accepted by one of them Turing Machine is halted whenever $P_A$ or $\overline{P_A}$ accepts *x*. It accepts all strings in *A* and rejects all strings in *A* so $P_A$ is decider for *A* and *A* is decidable.

As, for every input, Turing machine is halted for each and every input, then it can be said that $A \leq_m \overline{A}$ is decidable.

Comments (1)

## Problem

In the proof of Theorem 5.15, we modified the Turing machine M so that it never tries to move its head off the left hand end of the tape. Suppose that we did not make this modification to M. Modify the PCP construction to handle this case.

THEOREM **5.15**

PCP is undecidable.

## Step-by-step solution

**Step 1** of 1

**Given:**

In theorem 5.15 Turing Machine *M* is modified so that head is never moved to the left hand side of the tape. Now, suppose modification is not made to Turing Machine *M*.

**Proof:**

Here, it is considered that changes are not made to Turing Machine so users are supposed to modify the Post Correspondence Problem to handle this case. Modification of Post Correspondence Problem is done as follows:

It is assumed that changes are not made to the Turing Machine so modification of PCP is done. As it is also given in the question that modification is done in a way so it head is never moved to the left end of the tape. So assume a case where its head at the leftmost tape cell and attempts to move left to do so add dominos left further:

$$[\frac{\#qa}{\#rb}]$$

Here, for every $q,r \in Q$ and $a,b \in r$, where $\delta(q,a) = (r,b,L)$.

And then replace the first dominos in the following manner:

$$[\frac{\#}{\#\#q_0, w_1, w_2 \ldots w_n}]$$

Hence, this Approach should be used to handle the case where head tries to move to the left.

Comment

## Problem

Let

$$T = \{\langle M \rangle \mid M \text{ is a TM that accepts } w^{\mathcal{R}} \text{ whenever it accepts } w\}.$$

Show that T is undecidable.

## Step-by-step solution

**Step 1** of 4

**Consider the problem statement provided in the textbook.**

Comment

**Step 2** of 4

Let $T = \{ < M > \mid M \text{ is a TM that accepts } w^R \text{ whenever it accepts } w \}$.

• It is already known that $L = \{(w, M): w \textit{ is accepted } \text{by } M\}$ is undecidable.

• Assume that $T$ is decidable, then there must exist a TM by which $T$ can be decided. Let's say $P$ is the Turing Machine that decides $T$.

Comment

**Step 3** of 4

**For any input $(w, M)$, $M'$ can be constructed as follows:**

If $w = w^R$, simulate $M$ on $w$. The $\Sigma$ is the alphabet set of $M$ and let $a, b \notin M$.

Let $\Sigma \cup \{a, b\}$ be the alphabet set of $M'$. Then for input $ab$, $M'$ will reject all the other strings except $ab$.

Now, simulate $M$ on $w$.

• If $M$ accepts $w$, $M'$ rejects.

• If $M$ rejects $w$, $M'$ accepts.

**Claim:** $P$ accepts $M'$ iff $M$ accepts $w$.

**Proof:** If $P$ accepts $M'$. Since, $M'$ rejects all the other strings which include $ba$ also, then $M'$ rejects $ab$ which implies $M$ accepts $w$.

If $w$ is accepted by $M$, then $M'$ rejects $ab$. Since, $M'$ rejects all the other strings, $M'$ is accepted by $P$.

Now, construct a TM, $Q$ for $L$. Construct $M'$ on input $(w, M)$ and run $P$ on it. $Q$ accepts iff $P$ accepts.

This contradict the fact that $L$ is undecidable.

Comment

**Step 4** of 4

**Therefore, $T$ is undecidable. Hence Proved.**

Comments (2)

## Problem

Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape when it is run on input w. Formulate this problem as a language and show that it is undecidable.

## Step-by-step solution

**Step 1** of 2

Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape when it is run on input *w*.

Comment

**Step 2** of 2

The language that denotes the problem is,

$$L = \{< M, w > | \ M \text{ is a two-tape Turing machine writes a nonblank symbol on second tape when it is run on input } w\}$$

In order to check the decidability, show that Construct a TM $A_{TM}$ reduces to *L*. Use proof by contradiction method to prove decidability of the language. Assume TM *T* decides *L*. Construct a TM *B* that uses *T* to decide $A_{TM}$.

$B =$ "On input $< M, w >$:

1. Use M to construct the two-tape Turing machine $S$.
   $S$="On input $x$:
   1. Simulate $M$ using the first tape on input $x$.
   2. If M is accepted then write a nonblank symbol on the second tape."
2. Run $T$ on $< S, w >$ to check whether $S$ on input $w$ writes a nonblank symbol on second tape.
3. If $T$ accepts, $M$ accepts $w$ then accept. Otherewise, reject."

Thus, *T* decides $A_{TM}$ which is undecidable. Therefore, L is undecidable.

Comment

## Problem

Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.

## Step-by-step solution

**Step 1** of 1

**Undecidability of the Turing machine problems**

In the problem it is given that a single tape Turing machine writes a blank symbol over a non-blank symbol or not in the time of computation on any string which is provided as input. For showing the language generated by formulating the problem is undecidable following proof has been written:

$$K = \left\{ \langle M, w \rangle \middle| \begin{array}{l} M \text{ is a Turing machine for input w,} \\ \text{Writes a non blank symbol on second tape.} \end{array} \right\}$$

Now, define a two tape Turing machine $N_1$. This Turing machine takes a pair of $M$ and *w* or $\langle M, w \rangle$ as input. Here, $M$ is a Turing machine and w is a string which is passed as input to Turing machine. Turing machine $N_1$ is stimulating the Turing machine $M$, by using the first tape of machine $M$ on input $w$.

Turing machine $N_1$ will ignore the second tape of the Turing machine $M$ till end of the string does not come. If the Turing machine $M$ accepts the input string $w$ then the Turing machine $N_1$ will write a non-blank symbol on its second tape. If the simulation is halt then the Turing machine has stopped working.

Now assume $T$ decides *K*. Then, user create *T'* for deciding $A_{TM}$. When the input $\langle M, w \rangle$ is passed to the Turing machine, firstly, it simulates the Turing machine $N_1$ and after that it will run on the Turing machine $T$ on $N_1$. The language $K$ is accepted when $T$ accept it otherwise, vice versa.

So, Turing machine $T'$ is accepted if and only if when $T$ accepts. $T$ accepts the $\langle M_1, \langle M, w \rangle \rangle$ if and only if *M1* has written on second tape and $M$ accepts $w$. So, ATM is decided by $T'$. But, $A_{TM}$ cannot contain any decider. Here, in the above problem, for deciding *K*, any $T$ is not present so the language $K$ is undecidable.

Comment

## Problem

Consider the problem of determining whether a single-tape Turing machine ever writes a blank symbol over a nonblank symbol during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.

## Step-by-step solution

**Step 1** of 3

**Formulating the given problem as a language:**

$$L = \left\{ \langle M, w \rangle \middle| \begin{array}{l} M \text{ is a single tape Turing machine which writes a blank symbol} \\ \text{on non blank symbol while computing any input string} \end{array} \right\}$$

Comment

**Step 2** of 3

**Proving that the given problem is undecidable:**

By using contradiction, assume that the language $L$ is decidable. Suppose that $N$ is a decider for proving the decidability of the language $L$. A Turing machine *N* can be constructed as:

$N$ = "On Input $\langle M, s \rangle$

• Construct a Turing machine $A'$ now:

a. $A'$ writes # (a non-blank symbol) if *M* writes a blank symbol

b. Whenever $A'$ reads #, use the transitions specified by the blank symbols.

c. $A'$ Writes # on the tape before accepting and overwrites it with a blank symbol.

• Output of $A''$ will be input for decider $N$. If $N(\langle M', s \rangle)$ accepts, accept, otherwise reject.

Comments (2)

**Step 3** of 3

Now, the conclusion can be made that a blank symbol is written by $A'$ only when $A'$ takes the input $s$. That is, $N$ is a decider for $A_{TM}$ which is a contradiction. Hence, **the given problem is undecidable.**

Comment

## Problem

A ***useless state*** in a Turing machine is one that is never entered on any input string. Consider the problem of determining whether a Turing machine has any useless states. Formulate this problem as a language and show that it is undecidable.

## Step-by-step solution

**Step 1** of 1

**Undecidability of the Turing machine problem:**

• The given problem is defined as the following language:

$USELESS_{TM} = \{\langle T, q\rangle \mid q \text{ is a useless state in TM T}\}$.

• Show that $USELESS_{TM}$ is undecidable by reducing $E_{TM}$ to $USELESS_{TM}$ ,where $E_{TM} = \{\langle T1\rangle \mid \text{T1 is a TM and } L(T1) = \varnothing\}$.

• Using the Theorem 5.2. it is already proved that $E_{TM}$ is undecidable

• Suppose that $USELESS_{TM}$ is decidable and that *TM R* decides it.

• Note that for any Turing machine M with accept state $q_{accept}$, $q_{accept}$ is useless if and only if $L(T1) = \varnothing$.

• Accordingly, since *TM* R solves $USELESS_{TM}$ , *R* can be used to check if $q_{accept}$ is a useless state to decide $E_{TM}$ .

Specifically, below is a *TM S* that decides $E_{TM}$ by using the decider R for $USELESS_{TM}$ as a subroutine:

S = "On input $\langle T\rangle$ , where *M* is a *TM*:

1. Run TM R on input $\langle T, q_{accept}\rangle$ , where $q_{accept}$ is the accept state of *T*.

2. If *R* accepts, accept. If *R* rejects, reject."

However, since it is known $E_{TM}$ is undecidable and there cannot be a *TM* that decides $USELESS_{TM}$ .

Hence it is proved, that the given problem is undecidable.

Comments (5)

## Problem

Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left when its head is on the left-most tape cell. Formulate this problem as a language and show that it is undecidable.

## Step-by-step solution

**Step 1** of 2

Consider the problem of determining whether a Turing machine *M* on an input *w* ever attempts to move its head left when its head is on the left-most tape cell. This problem is formulated as a language:

$$L = \{< M, w > | \ M \text{ attempts to move its head left when its head is on the leftmost tape cell}\}$$

Comment

**Step 2** of 2

Assume that the language *L* is decidable and $\hat{M}$ be a TM that decides the language *L*. Construct a TM, *A* that decides the halting problem.

$A =$ "on input $<M, w>$:

1. Construct a TM, $A'$, from $A$. The TM $A'$ moves w one tape cell to the right and marks the leftmost cell with #.
2. Run the TM $A'$ on $<M, w>$.
3. If $A'$ encounters # then $A'$ moves to the right side and simulates $M$ reaching the leftmost tape cell.
4. If $M$ halts and accepts on $w$ then $A'$ simulates to move its head left when its head is on the leftmost tape cell."

Now, TM *A* runs $\hat{M}$ on the input $<A, w>$. If $\hat{M}$ accepts, *A* accepts. Otherwise, *A* rejects. It is assumed that $\hat{M}$ be a TM that decides the language *L*. If *M* halts and accepts on *w,* then only $A'$ moves its head left when its head is on the left-most tape cell. If *A* decides the halting problem, then halting problem is decidable. Thus, the halting problem is undecidable. It is a contradiction.

Therefore, the language *L* is undecidable.

Comments (1)

## Problem

Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left at any point during its computation on w. Formulate this problem as a language and show that it is decidable.

## Step-by-step solution

**Step 1** of 2

Consider a problem of determining whether a Turing machine *M* on input *w* ever attempts to move its head left at any point during its computation on *w*. The language that describes the problem is,

$$L = \{< M, w > | \ M \text{ moves its head left on input } w\}$$

Comment

**Step 2** of 2

Construct a Turing machine *A* that decides the problem.

$A =$ "On input $<M, w>$:

1. Run the machine for $|Q|+|w|+1$ steps.
2. If the Turing machine $M$'s head moved to the left then accept. Otherwise, reject."

Here, $|Q|$ represents the number of states and $|w|$ represents the length of the input string. The problem is said to be decidable, if the Turing machine *M* moves its head left on input *w* within the first $|Q|+|w|+1$ steps. The problem is decidable because, there exists a Turing machine for it.

**Therefore, the language *L* is decidable.**

Comments (1)

## Problem

Let $\Gamma = \{0, 1, \sqcup\}$ as follows. For each value of k, consider all k-state TMs that halt when started with a blank tape. Let *BB*(k) be the maximum number of 1s that remain on the tape among all of these machines. Show that *BB* is not a computable function.

## Step-by-step solution

**Step 1** of 1

**Given:** A tape of alphabets $\Gamma$ for all the Turing machines is given. $\Gamma$ Is composed of string of three terminals which is as follows: $\Gamma = \{0,1,\sqcup\}$

For every value of $k$ all $k$ states, of Turing machine halts, when the Turing machine starts from the blank tape. Maximum number of 1s is $BB(k)$ which remains in tape.

**Proof:**

By using contradiction, assume that the busy beaver function $BB$ is computable. If function $BB$ is computable then there exists a Turing machine $F$ for computing it. Now without loss of generality a Turing machine $F$ on input $1^n$, and the Turing machine $F$ halts with $1^{BB(n)}$ for each value of $n$.

Now build a Turing machine $M$, and this Turing machine halts when it will start from a blank tape based on F.

**Construction of Turing machine:**

Here $M$ is a Turing machine which halts when starts from blank tape.

**Step1:** Now the Turing machine $M$ writes n number of 1s on the tape.

**Step2:** Turing machine $M$ doubles the number of 1s on the tape.

**Step 3:** Now $M$ executes the Turing machine $F$ on the input $1^{2n}$.

Hence Turing machine $M$ will halts with $BB(2n)$ number of 1s if it is starts from the blank tape.

For implementing the Turing machine $M$, at most n numbers of states are required for the step 1 of the Turing machine and $c$ numbers of states are required for step 2 and 3, $c$ is a constant.

**Conclusion:**

By definition, $BB(n+c)$ is the maximum number of1s on which Turing machine with states $(n+c)$ will halt is at least the number of 1s on which the Turing machine $M$ halts. Which means $BB(n+c) \geq BB(2n)$ and this relationship will hold for all values of $n$.

Therefore, $BB(k)$ is strictly increasing function so $BB(n+c) < BB(2n)$. It proves wrong to our contradiction.

Hence, it is clear that $BB(k)$ is not a computable function.

Comment

## Problem

Show that the Post Correspondence Problem is decidable over the unary alphabet ∑ = {1}.

## Step-by-step solution

**Step 1** of 1

**Decidability of PCP over unary alphabet**

Post Correspondence Problem is basically concerned with manipulation of string and used to find match. Conceptually this concept is quite obvious by its statement itself as there is only 1 alphabet in the Post Correspondence Problem. PCP is decidable over the unary alphabet as follows:

Consider a Turing Machine M that runs on input $<P>$

1. Check for some $i$, $a_i$ is equals to $b_i$ then accept the string.

2. Check if $a_i > b_i$ or $a_j < b_j$ if there exist some $i$ and $j$ then accepts the string otherwise reject it.

For the first stage, Turing machine $M$ verifies for a domino which is forming a match. In the second stage Turing machine $M$ is looking for two dominos which are forming a match. If the Turing machine finds such pair then it builds a math by extracting $(b_j - a_j)$ copies of the $i^{th}$ dominos. After extracting copies the Turing machine puts them together with $(a_i - b_i)$ copies of the $j^{th}$ dominos.

This construction has $a_i(b_j - a_j) + a_j(a_i - b_i) = a_i b_j - a_j b_i$

1's on the top and $b_i(b_j - a_j) + b_j(a_i - b_i) = a_j b_i - a_j b_i$

1's on bottom. If any stage of Turing machine is not accepted then problem instance includes dominos with all upper parts having either more or less 1's than lower parts.

In this case, any match is not present hence Turing machine $M$ rejects.

Comment

## Problem

Show that the Post Correspondence Problem is undecidable over the binary alphabet ∑ = {0,1}.

## Step-by-step solution

**Step 1** of 1

Post Correspondence Problem is basically concerned with manipulation of string and used to find match.

Conceptually this concept is quite obvious by its statement itself as there are 2 alphabets in the Post Correspondence Problem. PCP is un-decidable over the two alphabets 0 and 1. This can be proved by using contradiction.

Assume the instance of PCP where the size of alphabet is *n* such that $n \le 2^m$, *m* is a constant. M-bit code is generated for the alphabets and then substituted in the tile.

Consider the tile in PCP given below:

$$\{[\frac{aba}{acd}],[\frac{aabd}{acd}],[\frac{a}{bc}],[\frac{acd}{bc}],[\frac{bcd}{abc}]\}$$

If the values of *a*, *b*, *c* and *d* is assigned as 00, 01, 10 and 11, the tiles given above will be written as:

$$\{[\frac{000100}{001011}],[\frac{00000111}{001011}],[\frac{00}{0110}],[\frac{001011}{0110}],[\frac{011011}{000110}]\}$$

The new PCP instance have alphabet size 2 will have solution if the original PCP instance have some solution. In the code, one-for-one substitution takes place. So, if the new instance has a solution, the solution will also exist for the original instance.

It should be taken care of the case when the new tiles provide a solution but the original tiles don't provide any solution. But this will never happens it is because if the tiles are obtained after inserting the values in the original instance.

For unary alphabets, the proof does not work. The codes are applied for *a*, *b*, *c* and *d* as 1, 11, 111 and 1111. If tracing is done backwards, it will be difficult to identify whether 1111 refers to *d, bb*, *ac* or *ca*(there will be other possibilities too). So, PCP is un-decidable.

Comment

## Problem

In the *silly Post Correspondence Problem*, *SPCP*, the top string in each pair has the same length as the bottom string. Show that the *SPCP* is decidable.

## Step-by-step solution

**Step 1** of 2

In silly Post correspondence problem or SPCP, each pair of both strings (top and bottom) have same length. Here, the concern is related to whether the instance contains a match or not. SPCP problem is decidable. Decidability of SPCP can be proved as follows:

Consider an instance of SPCP given below:

$$\{[\frac{a_1}{b_1}], [\frac{a_2}{b_2}], \ldots\ldots, [\frac{a_n}{b_n}]\}$$

Here, $|a_i| = |b_k|$ for all $1 \le i \le k$.

Comment

**Step 2** of 2

After finding the match of SPCP instance, it is checked whether the numerator is equal to denominator or not for the domino. Here, the length of the strings at top and bottom are similar.

Trivial match in a single domino of SPCP is formed whenever the top and bottom string is same.

After examining the instance having same top and bottom string, it is said to be as decidable. Dominos are working as decider to find whether the top string and bottom string are same or not. This it is quite easy to find decidability of SPCP.

Comment

## Problem

Prove that there exists an undecidable subset of $\{1\}^*$.

## Step-by-step solution

**Step 1** of 2

Un-decidable subset of $\{1\}^*$ exists. Existence of un-decidable subset can be found as follows:

Consider there is an $i^{th}$ position in any string generated from $\{1\}^*$. It is quite obvious that there are uncountable number of strings can be made using $\{1\}^*$.

Comment

**Step 2** of 2

Let $S$ be the subset of $\{1\}^*$. The $i^{th}$ position of the generated string would be 1 if the generated string is in $S$. The number of subsets of $\{1\}^*$ is equal to the infinite number of binary strings which is uncountable. The uncountable set is not Turing recognizable because there is more number of subsets than the number of Turing machines.

Therefore, as it is proved above it is not recognizable so it will be un-decidable.

Comment

## Problem

Let $AMBIG_{CFG}$ = $\{\langle G\rangle \mid G \text{ is an ambiguous CFG}\}.$ Show that *AMBIG*CFG is undecidable. (Hint: Use a reduction from *PCP*. Given an instance

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\}$$

of the Post Correspondence Problem, construct a CFG G with the rules

$$\begin{aligned} S &\to T \mid B \\ T &\to t_1 T \mathtt{a}_1 \mid \cdots \mid t_k T \mathtt{a}_k \mid t_1 \mathtt{a}_1 \mid \cdots \mid t_k \mathtt{a}_k \\ B &\to b_1 B \mathtt{a}_1 \mid \cdots \mid b_k B \mathtt{a}_k \mid b_1 \mathtt{a}_1 \mid \cdots \mid b_k \mathtt{a}_k , \end{aligned}$$

where $a_1, \ldots, a_k$ are new terminal symbols. Prove that this reduction works.)

## Step-by-step solution

**Step 1** of 1

**Un-decidability**

1. If $P$ has match with $t_{i1}t_{i2}\cdots t_{il} = b_{i1}b_{i2}\cdots b_{il}$ then it can be observed that string $t_{i1}t_{i2}\cdots t_{il}a_{il}\cdots a_{i2}a_{i1}$ has minimum two derivations, first from $T$ and other one from $B$.

2. If the Context free grammar $G$ is ambiguous, then some string $s$ should have multiple derivations. As $G$ generate $s$, $s$ can be written as $wa_{j1}a_{j2}\cdots a_{jm}$ for some $w$ that do not have symbols from $a_i's$.

After checking the grammar $G$, It can be observe that the derivation of $B$ and derivation of $T$ can each generate maximum one strings of same form as $s$. The multiple derivations of $s$ as follows:

$$S \Rightarrow T \overset{*}{\Rightarrow} s = t_{jm}t_{jm-1}\cdots t_{j1}a_{j1}a_{j2}\cdots a_{jm}$$
$$S \Rightarrow B \overset{*}{\Rightarrow} s = b_{jm}b_{jm-1}\cdots b_{j1}a_{j1}a_{j2}\cdots a_{jm}$$

Thus, $t_{jm}t_{jm-1}\cdots t_{j1} = b_{jm}b_{jm-1}\cdots b_{j1}$

By combining $(1)$ and $(2)$, P has a match iff $G$ is ambiguous.

So, the reduction from $PCP$ to $AMBIG_{CFG}$ works. Thus, $AMBIG_{CFG}$ is un-decidable.

Comment

## Problem

Show that A is Turing-recognizable iff $A \leq_m A_{TM}$.

## Step-by-step solution

**Step 1** of 4

**Turing Machine:**

• This is a type of mathematical model of computation that specify the conceptual machine.

• The Turing machine that operate a strings symbols on the tape as per given set of rules.

• The Turing machine can be manufactured in order to imitate the algorithms.

Comment

**Step 2** of 4

**Turing Recognizable:**

• Any language will be recognizable or not if and only if it depends on Turing machine which will stop and receive only strings in that language and for strings not in the language.

• So, the Turing machine then refuse or does not stop at all.

Comment

**Step 3** of 4

**Theorem of Turing Recognizable:**

• As per given in the theorem 5.28 from the book.

• If $A <=_m A_{TM}$ in this $A_{TM}$ is Turing Recognizable then $A$ is Turing Recognizable.

• $A$ is Turing recognizable that can be prove below which is as follows:

Comment

**Step 4** of 4

**Consider the following details which is as follows:**

**A is Turing Recognizable iff $A <=_m A_{TM}$ .**

• If $A <=_m A_{TM}$ that means then A is Turing recognizable since $A_{TM}$ is Turing recognizable.

• Suppose that A is Turing Recognizable then there will exists a Turing machine T that identify A.

• That means T gets the inputs which is $w$ and receive if $w \in A$ that means $w$ belongs to $A$.

• Apart from that T does not receive.

• In order to exhibit the $A <=_m A_{TM}$.

• Describe a Turing machine that does the following works which is as follows:

• Take input $w$ and write the $(T, w)$ on the tape and stop.

• Then check the $(T, w)$ is in $A_{TM}$.

• The above condition can be checked if and only if $w$ is in $A$.

- So, get the mapping reduction of $A$ to $A_{TM}$.
- So, the reduction proofs that A is Turing Recognizable.

**Hence, it is proved that A is Turing Recognizable.**

Comments (4)

## Problem

Show that A is decidable iff A $\leq_m 0^*1^*$.

## Step-by-step solution

**Step 1** of 3

**Decidability**

Assume $B = 0^*1^*$. Thus it is required to prove that $A$ is decidable iff $A \leq_m B$.

Solution can be divided into two parts.

**1:** If $A$ is decidable then $A \leq_m B$.

**2:** If $A \leq_m B$ then $A$ is decidable.

Comment

**Step 2** of 3

**Part 1:** If $A$ is decidable then $A \leq_m B$.

Proof: Firstly define a function $f$ as follows:

$$f(s) = 01 \text{ if } s \in A$$
$$f(s) = 10 \text{ otherwise.}$$

Since $A$ is decidable, decider can be used for $A$ to compute $f$. Also, $s \in A \text{ iff } f(s) \in B.$

Hence, $f$ is mapping reduction from $A$ to $B$.

Comment

**Step 3** of 3

**Part 2:** If $A \leq_m B$, then $A$ is decidable.

Proof: Since $A \leq_m B$, there exist a function $f$, such that $w \in A$ iff $f(w) \in B$.

Now consider Turing Machine M:

M= On input $w$

1. Compute $f(w)$

2. If $f(w)$ is in form of $0^*1^*$, then accept, Otherwise, reject.

Now

$$\begin{aligned} & w \in A \\ & \Leftrightarrow f(w) \text{ is of the form } 0^*1^* \\ & \Leftrightarrow M \text{ accept } w. \end{aligned}$$

Thus, $M$ decides $A$.

Comment

## Problem

Let J =

$$\{w|\text{ either } w = 0x \text{ for some } x \in A_{\mathsf{TM}}, \text{ or } w = 1y \text{ for some } y \in \overline{A_{\mathsf{TM}}}\ \}.$$

is Turing-recognizable.

## Step-by-step solution

**Step 1** of 2

**Turing-recognizable**

Firstly demonstrate the reduction $f:\Sigma^* \to \Sigma^*$ of $\overline{A_{TM}}$ to $J$.

Assume a string $z \in \Sigma^*$. So that $f(z) = 1z$.

By definition of $J$, $z \in \overline{A_{TM}}$ iff $1z \in J$.

Hence $f$ is a reduction of $\overline{A_{TM}}$ to $J$, Thus $\overline{A_{TM}} \leq_m J$.

By using the Corollary:

**"If $\overline{A_{TM}} \leq_m B$, $A$ is not a Turing-recognizable, then $B$ is not Turing-recognizable."**

Because $\overline{A_{TM}}$ is not Turing-recognizable, by Corollary $J$ is not Turing-recognizable.

Comment

**Step 2** of 2

Now demonstrate the reduction $f:\Sigma^* \to \Sigma^*$ of $A_{TM}$ to $J$.

Assume a string $t \in \Sigma^*$. So that $g(t) = 0t$.

By definition of $J$, $t \in A_{TM}$ iff $0t \in J$.

Hence $g$ is reduction of $A_{TM}$ to $J$, Thus $A_{TM} \leq_m J$.

A function which reduces language $L_1$ to language $L_2$ also reduces $\overline{L_1}$ to language $\overline{L_2}$. Hence, $g$ is reduction from $\overline{A_{TM}}$ to $\overline{J}$, Thus $\overline{A_{TM}} \leq_m \overline{J}$.

By using the Corollary:

**"If $\overline{A_{TM}} \leq_m B$, $A$ is not a Turing-recognizable, then $B$ is not Turing-recognizable."**

Because $\overline{A_{TM}}$ is not Turing-recognizable, by Corollary $\overline{J}$ is also not Turing-recognizable.

Therefore neither $J$ nor $\overline{J}$ is Turing-recognizable.

Comment

## Problem

Give an example of an undecidable language $B$, where $B \leq_{\mathrm{m}} \overline{B}$.

## Step-by-step solution

**Step 1** of 2

**Undecidable language:**

A language is an undecidable language, if it is not Turing-decidable. In other words, a language is undecidable language when there exists no Turing machine that can decide the language.

For example, let $B_{TM} = \{\langle M, w\rangle \mid M \text{ is a } TM \text{ and accepts the input '} w \text{'}\}$ is undecidable.

Comments (1)

**Step 2** of 2

**Proof by contradiction:**

Assume that $B_{TM}$ is decidable.

Assume that the Turing machine $A$ decides $B_{TM}$. So, the decidabilty of the Turing machine *A* is defined as:

$$A\langle M, w\rangle = \begin{cases} accept & \text{if } M \text{ accepts input } w \\ reject & \text{if } M \text{ does not accept the input } w \end{cases}$$

Using the Turing machine *A*, construct another Turing machine *X* that decides whether a machine *M* accepts its own encoding $\langle M\rangle$ is:

1. Input is $\langle M\rangle$, where M is some Turing machine.

2. Run *A* on $\langle M, \langle M\rangle\rangle$.

3. If *A* accepts the language, *reject*. Otherwise, *accept*.

So, the decidabilty of the Turing machine *X* is defined as:

$$X\langle M\rangle = \begin{cases} accept & \text{if } M \text{ does not accept } \langle M\rangle \\ reject & \text{if } M \text{ accepts } \langle M\rangle \end{cases}$$

The above specification cannot be satisfied by the machine. The Turing decidability of X on its own encoding $\langle X\rangle$ is:

$$X\langle X\rangle = \begin{cases} accept & \text{if } D \text{ does not accept } \langle D\rangle \\ reject & \text{if } D \text{ accepts } \langle D\rangle \end{cases}$$

Hence, neither *X* nor *A* can exist. That is, neither *X* nor *A* can accept the Turing machine *M*.

**Thus, $B_{TM}$ is undecidable.**

Comments (7)

## Problem

Define a ***two-headed finite automaton*** (2DFA) to be a deterministic finite automaton that has two read-only, bidirectional heads that start at the left-hand end of the input tape and can be independently controlled to move in either direction. The tape of a 2DFA is finite and is just large enough to contain the input plus two additional blank tape cells, one on the left-hand end and one on the right-hand end, that serve as delimiters. A 2DFA accepts its input by entering a special accept state. For example, a 2DFA can recognize the language $\{a^n b^n c^n \mid n \geq 0\}$.

**a.** Let $A_{\text{2DFA}} = \{\langle M, x\rangle \mid M \text{ is a 2DFA and } M \text{ accepts } x\}.$ Show that $A_{2DFA}$ is decidable.

**b.** Let $E_{\text{2DFA}} = \{\langle M\rangle \mid M \text{ is a 2DFA and } L(M) = \emptyset\}.$ Show that $E_{2DFA}$ is not decidable.

## Step-by-step solution

**Step 1** of 2

**Decidability of 2DFA**

Consider a Turing machine to check whether *M* the 2DFA accept the input *x*. if any configuration is repeated in M then it will not terminate because it is a deterministic finite automata.

Consider a Turing machine *W* which encodes M the 2DFA and input *x*. It also simulate M on *x* and check *M* accepts *x*. *W* has four tapes.

• Input tape to store input.

• Work tape, which has two bidirectional head to read.

• Another work tape to store configuration occurs during simulation.

• Scratch tape used to create representation of *M* configuration which becomes helpful for work tape in searching and updating.

*W* Turing machine works as follow:

$W = \text{on input } \langle M, x\rangle, \text{where } M \text{ is a 2DFA and } x \text{ is a string}$

1. Check the input tape $\langle M, x\rangle$ has a proper legal encoding or not. If it does not contain legal coding, then reject and halt, otherwise continue.

2. Copy input to work tape. Initialize the two head of work tape so that they are their starting position. Also initialize the second work tape as empty.

3. When *M* current state has halt then accepts and halt the states. When *M* current state has no move then reject and halt the states.

4. Create configuration on scratch tape. When current configuration already exist in the work tape then reject and halt otherwise store configuration at the end of second work tape.

5. Simulate one move of *M* on input tape.

6. Move to step 3.

When the input *x* are accepted by 2DFA that is M, the simulation will completed this in finite number of steps, then *W* will accept the input. Otherwise, when input codes are not legal or M does not ends or terminates. *W* determines this in finite number of steps and rejects the input.

All this shows that language $A_{2DFA}$ is decidable.

Comment

**Step 2** of 2

Assume on contrary that $E_{2DFA}$ is decidable. Consider *W* a decider Turing machine which decides the $E_{2DFA}$.

Now, create Turing machine E which is based on *W* for deciding $E_{TM}$ which works as follow:

$E = \text{on input } \langle M\rangle$

Create another 2DFA *M'*. The accepting computation history of *M* is recognizing by this *M'*.

Execute *W* on *M'*. When *W* accepts, **accepts**. Else **reject**.

Since, the $E_{TM}$ is not decidable therefore the contradiction occurs. Hence $E_{2DFA}$ is undecidable.

## Problem

Next

A ***two-dimensional finite automaton*** (2DIM-DFA) is defined as follows. The input is an m × n rectangle, for any m, n ≥ 2. The squares along the boundary of the rectangle contain the symbol # and the internal squares contain symbols over the input alphabet ∑. The transition function $\delta: Q \times (\Sigma \cup \{\#\}) \longrightarrow Q \times \{\mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{D}\}$ indicates the next state and the new head position (Left, Right, Up, Down). The machine accepts when it enters one of the designated accept states. It rejects if it tries to move off the input rectangle or if it never halts. Two such machines are equivalent if they accept the same rectangles. Consider the problem of determining whether two of these machines are equivalent. Formulate this problem as a language and show that it is undecidable.

## Step-by-step solution

**Step 1** of 2

To check **the un-decidability** of $E_{2DIM-DFA}$, first of all, it will have to $m-reduce$ $A_{TM}$ to $E_{2DIM-DFA}$ by using a mapping which takes $\langle M, w \rangle$ **to** $\langle B \rangle$. It means that if $w$ **is accepted by** $M$ **then** $L(B)$ **is desired to be non-empty and when** $w$ **is not accepted by** $M$ **then** $L(B)$ **is desired to be empty.**

Comment

**Step 2** of 2

This can be achieved by making $L(B)$ be the set of accepting, all the histories of $M$ on $w$ Here, $B$ can be used as a rectangle (to represent the computation history of $C_1, C_2, \ldots, C_k$) with $C_1$ in the first row and $C_2$ in the next row and so on.

• For a given input rectangle, checking is performed. Here, $B$ checks that the initial configuration of $M$ on $w$ is in first row or not and the last row is an accepting or final configuration .

• It also checked that each rows is followed from a previous row by the given rules of $M$ .

• Now, if $w$ is accepted by $M$ , then there exists an accepting configuration history of $M$ on $w$ and $L(B)$ is not equals to $\varnothing$ or $L(B) \neq \varnothing$.

• In the same way, $L(B) = \varnothing$ and there is no accepting configuration history of $M$ on $w$, when $w$ is not accepted by $M$ . Thus, $m-reduce$ $A_{TM}$ to $\overline{E}_{2DIM-DFA}$ and $E_{2DIM-DFA}$ is un-decidable.

**So, It follows that** $EQ_{2DIM-DFA}$ **is un-decidable.**

Comment

## Problem

**Rice's theorem**. Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turingmachine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language whenever $L(M_1) = L(M_2)$, we have

$$\langle M_1 \rangle \in P \text{ iff } \langle M_2 \rangle \in P.$$

Here, $M_1$ and $M_2$ are any TMs. Prove that P is an undecidable language.

## Step-by-step solution

**Step 1** of 1

**Rice's theorem**

Given $P$ non trivial property of language of Turing machine, it is required to prove that P is un-decidable. Consider on the contrary that $P$ is decidable language that satisfies the properties. Consider $R_P$ be a Turing machine that decides $P$. Now it is required to show that how to decide $A_{TM}$ using $R_P$ by constructing Turing machine $S$.

First, let $T_\theta$ be a Turing machine that always reject, so $L(T_\theta) = \phi$. It can be consider that $\langle T_\theta \rangle \notin P$ without loss of generality, because it is possible to proceed with $\overline{P}$ instead of $P$ if $\langle T_\theta \rangle \in P$. Because $P$ is non-trivial, there exist a Turing machine $T$ with $\langle T \rangle \in P$. Now construct $S$ based on $T$ and $R_P$ as follows:

$S =''$ On input $\langle \mathrm{M,w} \rangle$:
1. Use $M$ and $w$ to construct the following Turing machine $M_w$.

$M_w =''$ On input $x$:
1. Simulate $M$ on $w$. If it halts and rejects,reject.
2. Simulate $T$ on $x$. If $T$ accepts $x$, accept."

2. Use TM $R_p$ to determine if $\langle M_w \rangle \in P$. If YES, accept,else reject."

Note that TM $M_w$ has property that $(1)$ if $M$ accept $w$, $L(M_w) = L(T)$, and $(2)$ if $M$ does not accept $w$, $L(M) = \phi = L(T_\phi)$.

In other words, $\langle M_w \rangle \in P$ if and only if $M$ accept $w$.

Since the construction of $M_w$ from $T$, $M$ and $w$ takes finite steps, the TM $S$ is decider for $A_{TM}$. This creates a contradiction since $A_{TM}$ is an un-decidable language. In conclusion, $P$ is un-decidable.

Comment

## Problem

Show that both conditions in Problem 5.28 are necessary for proving that P is undecidable.

Problem 5.28

**Rice's theorem**. Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$. Here, $M_1$ and $M_2$ are any TMs. Prove that P is an undecidable language.

## Step-by-step solution

**Step 1** of 1

**Condition to show that some language un-decidable**

First, consider $P$ be the language $\{\langle M \rangle \mid M \text{ is a Turing Machine with 5 states}\}$. $P$ is non-trivial, and so it satisfies the second condition of Rice's Theorem but $P$ can be easily decided by checking the number of states of the input Turing Machine.

Second, consider $P$ be the empty set. Then, it does not contain any Turing Machine and so, it satisfies the first condition of Rice's Theorem, but $P$ can be decided by a Turing Machine that always rejects. Hence, both the properties are required for proving $P$ un-decidable.

Comment

## Problem

Use Rice's theorem, which appears in Problem 5.28, to prove the undecidability of each of the following languages.

A **a.** $INFINITE_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a } \mathsf{TM} \text{ and } L(M) \text{ is an infinite language}\}$.

**b.** $\{\langle M\rangle \mid M \text{ is a } \mathsf{TM} \text{ and } 1011 \in L(M)\}$.

**c.** $ALL_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a } \mathsf{TM} \text{ and } L(M) = \Sigma^*\}$.

Problem 5.28

**Rice's theorem**. Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language whenever $L(M_1) = L(M_2)$, we have $\langle M_1\rangle \in P$ iff $\langle M_2\rangle \in P$. Here, $M_1$ and $M_2$ are any TMs. Prove that P is an undecidable language.

## Step-by-step solution

**Step 1** of 3

**Un-decidability using Rice Theorem**

a. $INFINITE_{TM}$ is a language of TM descriptions. It satisfies the two conditions of Rice's Theorem. First, is that it is non-trivial because some TMs have infinite languages and others have not. Second, is that it depends only on language. If two Turing Machine recognize same language, then either both should have descriptions in $INFINITE_{TM}$ or neither does. Consequently, Rice's theorem implies that $INFINITE_{TM}$ is un-decidable.

Comment

**Step 2** of 3

b. Consider $\{\langle M\rangle \mid M \text{ is a Turing Machine and } 1011 \in L(M)\}$. $P$ is language of Turing Machine descriptions. It satisfies the two conditions of Rice's Theorem. First, it is non-trivial because some TMs contain the string 1011 in their language and others do not. Second, it only depends on the language. If two TMs recognize same language then either both should have descriptions in $P$ (because they both accept 1011), or neither do. Thus, Rice's theorem implies that $P$ is un-decidable.

Comments (1)

**Step 3** of 3

c. $ALL_{TM}$ is a language of TM descriptions. It satisfies the two conditions of Rice's Theorem. First, it is non-trivial because some TMs accept all possible strings of an alphabet $\Sigma$ and others do not. Second, is that it depends only on language. If two TMs recognize same language, then either both should have descriptions in $ALL_{TM}$ or neither does. Therefore, Rice's theorem implies that $ALL_{TM}$ is un-decidable.

Comment

## Problem

Let

$$f(x) = \begin{cases} 3x+1 & \text{for odd } x \\ x/2 & \text{for even } x \end{cases}$$

for any natural number x. If you start with an integer x and iterate f, you obtain a sequence, x, f(x), f(f(x)), . . . . Stop if you ever hit 1. For example, if x = 17, you get the sequence 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Extensive computer tests have shown that every starting point between 1 and a large positive integer gives a sequence that ends in 1. But the question of whether all positive starting points end up at 1 is unsolved; it is called the $3_{x+1}$ problem.

Suppose that $A_{TM}$ were decidable by a TM H. Use H to describe a TM that is guaranteed to state the answer to the 3x + 1 problem.

## Step-by-step solution

**Step 1** of 1

The solution can be built in stages. Later stages will use Turing machines that are constructed in earlier stages. The first step is to construct a Turing machine $M_{\text{query}}$ that takes an input $x$ and accepts if iterating $f$ starting from $x$ eventually yields 1 and loops forever otherwise.

$M_{\text{query}}$. On input $\langle x \rangle$:

1. If $x = 1$, then accept.
2. If $x$ is odd, update $x \leftarrow 3x+1$. If $x$ is even, update $x \leftarrow \frac{x}{2}$.
3. Go to step 1.

Our next step is to construct a Turing machine $M_{\text{loop}}$ which iterates over all positive integers, looking for a counter-example to the $3x+1$ conjecture. That is, $M_{\text{loop}}$ searches for some $x$ such that iterating $f$ starting from $x$ never reaches $1$ and accepts if it finds such an $x$. To do this, it seems tempting to have $M_{\text{loop}}$ simulated $M_{\text{query}}$ first on $x=1$, next on $x=2$, and so forth. Whenever iterating $f$ on $x$ yields $1$, the simulation of $M_{\text{query}}$ will eventually end and $M_{\text{loop}}$ would then proceed to the next number $x+1$. But what if $M_{\text{loop}}$ actually finds a counter-example $x$? In this case, the simulation of $M_{\text{query}}$ on $x$ will never terminate, and $M_{\text{loop}}$ will be in the unfortunate situation that it has found what it is looking for, but it doesn't know it has found it!

To get around this, use $H$. Instead of simulating $M_{\text{query}}$ on $x$, It has $M_{\text{loop}}$ check whether or not $M_{\text{query}}$ would accept $x$ by passing $\langle M_{query}, x \rangle$, to $H$.

$M_{\text{loop}}$. On input $\langle w \rangle$:

1. Ignore the input $w$.
2. For each natural number $y = 1, 2, 3, \ldots$:
3. Run $H$ on $\langle M_{\text{query}}, y \rangle$.
4. If $H$ rejects, then accept. Otherwise, continue the loop.

Finally, in order to solve the $3x+1$ problem, it is required to know whether or not $M_{\text{loop}}$ finds a counter-example. Again, it might be tempted to simulate $M_{\text{loop}}$ and see if it ever finds a counter-example and accepts. The problem is that there may not be any counter-example, in which case $M_{\text{loop}}$ will loop forever and our simulation will not terminate. The trick is to use $H$ again to see if $M_{\text{loop}}$ finds a counter example.

$M_{3x+1.}$ On input $\langle \text{w} \rangle$:

1. Ignore the input $w$.
2. Run $H$ on $\langle M_{\text{loop}}, \varepsilon \rangle$.
3. If $H$ accepts, then print ="There is a counter-example to the $3x+1$ conjecture."

Comment

## Problem

Prove that the following two languages are undecidable.

**a.** $OVERLAP_{\mathsf{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs where } L(G) \cap L(H) \neq \emptyset\}$. (Hint: Adapt the hint in Problem 5.21.)

**b.** $PREFIX\text{-}FREE_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG where } L(G) \text{ is prefix-free}\}$.

## Step-by-step solution

**Step 1** of 7

The given languages have to be proven to be un-decidable.

a)

$$OVERLAP_{CFG} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs where } L(G) \cap L(H) \neq \phi\}$$

Comment

**Step 2** of 7

Assume that $OVERLAP_{CFG}$ is decidable. Given an instance for the problem of Post Correspondence $P = \left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_n}{b_n}\right]\right\}$, introduce unique new terminals $a_1, a_2, \ldots, a_n$ for the CFGs.

Comment

**Step 3** of 7

• Define the CFG $G$ as:

$$G = t_1, t_2, \ldots, t_n$$
$$L(G) = \{s \mid s = t_i t_j \ldots t_k a_k \ldots a_j a_i\}$$
$$S_G \to t_1 S_G a_1 \mid \ldots \mid t_n S_G a_n \mid t_1 a_1 \mid \ldots \mid t_n a_n$$

• Similarly, define the CFG $H$ as follows:

$$H = b_1, b_2, \ldots, b_n$$
$$L(H) = \{s \mid s = b_i b_j \ldots b_k a_k \ldots a_j a_i\}$$
$$S_H \to b_1 S_H a_1 \mid \ldots \mid b_n S_H a_n \mid b_1 a_1 \mid \ldots \mid b_n a_n$$

As $L(G) \cap L(H) \neq \phi$, we get $t_i t_j \ldots t_k a_k \ldots a_j a_i = b_i b_j \ldots b_k a_k \ldots a_j a_i$.

Comment

**Step 4** of 7

• Since the new terminals $a_1, a_2, \ldots, a_n$ are unique, which can be cancelled from both sides resulting in:

$$t_i t_j \ldots t_k = b_i b_j \ldots b_k.$$

This is a way to solve for **the Post Correspondence Problem** $P$**.** This is a contradiction as the Post Correspondence **Problem is un-decidable**. Therefore, the assumption taken that $OVERLAP_{CFG}$ **is decidable, is incorrect**.

Comment

**Step 5** of 7

b)

$PREFIX\text{-}FREE_{CFG} = \{\langle G\rangle \mid G \text{ is a CFG where } L(G) \text{ is prefix-free}\}$

A mapping reducibility from the language $A_{TM}$ (does a Turing machine accept a string?) to the language $PREFIX\text{-}FREE_{CFG}$ is given by the function $f$. Here, $\langle M, w\rangle$ is taken as an input of the computable functions $f$ and $\langle M', w'\rangle$ is returned in such a way that:

$\langle M, w\rangle \in A_{TM}$ iff $\langle M', w'\rangle \in PREFIX\text{-}FREE_{CFG}$

Comment

**Step 6** of 7

The machine $F$ to compute the function $f$ is:

$F =$"when $\langle M, w\rangle$ is taken as an input:

1. The machine $M'$ is constructed

$M' =$"On input $x$:

1. For all proper prefixes $y$ of $x$:

1. $M$ will be run on $y$.

2. then *reject*.

2. Run $M$ on $x$.

3. If it is accepted by $M$, then *accept*.

4. If it is rejected by $M$,, then *reject*."

2. Output $\langle M', w'\rangle$.

The output machine $M'$ only accepts an input string $w$ if the language $L(M)$ is prefix-free. It does so by checking if any of the proper prefixes of $w$ do not lie in $L(M)$ and $w$ lies in it.

Comment

**Step 7** of 7

• It has been shown that $A_{TM}$ **is mapping reducible to** $PREFIX\text{-FREE}_{CFG}$, that is:

$A_{TM} \leq_m PREFIX\text{-}FREE_{CFG}$

**Thus as** $A_{TM}$ **is un-decidable, the language** $PREFIX\text{-FREE}_{CFG}$ **is also un-decidable.**

Comment

## Problem

Consider the problem of determining whether a PDA accepts some string of the form {wwl w ? $\{0,1\}^*$} . Use the computation history method to show that this problem is undecidable.

## Step-by-step solution

**Step 1** of 1

When $M$ does not accept $w$ , then $M_{new}$ will not go to left of $\$$. The Turing machine $M_{new}$ writes something on the original input when $M$ accept $w$ .

Consider a Turing machine *T* having input *x*. Create a PDA which accepts string in $h\$h\$$ form when $x \in L(T)$.

The description history of *T* for *x* is the configuration sequence $C_0, C_1, C_2 \ldots C_n$ where $C_0$ is the starting configuration of *T* in input *x*. The configuration $C_n$ is the end or accepting configuration of *T* in input *x*.

Working of Turing machine:

1. At first PDA insert $C_0$ into stack and then confirms whether $C_0$, is the initial configuration in Turing machine T on input *w*.

2. At each odd configuration the PDA pops the last configuration from stack and then checks that the new configuration is the successor of last one or not.

3. When PDA read and first $ encountered, then it skips the $C_0$ configuration and insert the $C_1$ in the stack. At this time, it has confirmation that even configuration is the successor of odd configuration.

4. The PDA gets ACCEPTS when all check are true which means all new configuration is the successor of previous one and last configuration is the accepting configuration.

The PDA accepts the string $ww$ if *x* is accepted by Turing machine *T*.

When *x* is not accepted by Turing machine *T* even then this PDA can accept the string in $y\$z\$$ form where $y \neq z$. This all happens because the construction depends on the different check whether input is in $ww$ form or which does not happen in PDA to confirm that two passes check the same history.

Hence, PDA accepts string is un-decidable.

Comments (3)

## Problem

Let X = { $\langle M, w \rangle |$ _ M is a single-tape TM that never modifies the portion of the tape that contains the input w}. Is X decidable? Prove your answer.

## Step-by-step solution

**Step 1** of 1

*X* is un-decidable which can be proof by reducibility. Reduce *X* from $D_{TM} = \{< M, w >| \text{ Turing machine } M \text{ accepts } w\}$.

Assume *W* be a Turing machine which decides *X*. Use Turing machine *W* to create Turing machine *S* which decides $D_{TM}$.

1. TM *S*: on input $< M, w >$

2. TM $M_{\text{new}}$: on input $< M, w >$

3. The right end of the input mark with symbol $\$ \notin \Gamma_{\text{M}}$.

4. Copy the string which comes after $. This parts of input denotes as $w'$

5. Simulate $M$ on $w'$.

6. If $M$ accepts, then write all strings written in first cell of input tape, **accept**

7. Else **reject.**

8. TM *W*: on input $< M_{new}, w >$

9. When *W* accepts, then **accept**, else **reject**.

When $M$ does not accept $w$, then $M_{new}$ will not go to left of $\$$. The Turing machine $M_{new}$ writes something on the original input when $M$ accept $w$.

It is shown above that $M_{new}$ modifies whenever *M* accepts the input *w*. Hence, $D_{TM}$ has a contradiction.

Comments (3)

## Problem

Say that a variable A in CFG G is ***necessary*** if it appears in every derivation of some string w ? G. Let $NECESSARY_{CFG}$ = { $\langle G, A \rangle |$ A is a necessary variable in G}.

**a.** Show that $NECESSARY_{CFG}$ is Turing-recognizable.

**b.** Show that $NECESSARY_{CFG}$ is undecidable.

## Step-by-step solution

**Step 1** of 2

Consider a CFG *G* which contains variables *A* in his each production of some string *w* which belongs to *G*.

Consider a Turing machine *D* which works as follow:

- $D = \text{On input} \langle G, A \rangle, \text{ where } G \text{ is CFG and } A \text{ is non terminal}$

1. Create context free grammar $\frac{G}{A}$ by eliminating variable A from the derivations of *G*.

2. Create list of strings *w* generated by grammar G. Create a decider for $A_{CFG}$ and then check each string of *w* can also be generated by $\frac{G}{A}$.

3. If *w* strings cannot generated by $\frac{G}{A}$ then **accept**, else **continue**.

• *D*= On other input instead of $\langle G, A \rangle$, **reject**

The $L\left(\frac{G}{A}\right)$ language contains all and only that strings of $w \in L(G)$ which does not require non terminal *A* for their derivation. If variable *A* is necessary for grammar *G* then few strings of $w \in L(G)$ cannot produce without use of non-terminal *A*.

The Turing machine *D* finds out those strings of *w* which cannot derived without use of *A*.

On the other hand, when variable A is not necessary for grammar G, which means $L(G) = L\left(\frac{G}{A}\right)$, then *D* continue move in a loop.

Hence, $\text{NECESSARY}_{CFG}$ recognize by Turing machine *D*.

Comment

**Step 2** of 2

It is already known that $ALL_{CFG}$ is un-decidable, this shows that $\text{NECESSARY}_{CFG}$ is also un-decidable. Therefore, $\text{NECESSARY}_{CFG}$ must also un-decidable.

When any language complement is un-decidable then that language will also become un-decidable.

This can be proved by reduction R. The computation of reduction R is as follow:

$\text{R} = \text{On input} \langle G \rangle :$

• Create production of *G* after adding variable A and their productions in G.

$S \to A$

$A \to \in$

And

$A \to aA$ for each $a \in \Sigma$

• The output $\langle G, A \rangle$

The Grammar *G* produce with the help of reduction *R* is always $L(G) = \Sigma_*$. Thus, if $L(G) = \Sigma_*$, then there is no need of variable *A* in *G* because each string of $w \in \Sigma_*$ can derived by G without using *A*.

Also, if $L(G) = \Sigma_*$ then variable *A* is necessary for production of *G* because if $w \in L(G)$ then production of *G* is only possible with the help of *A*.

Thus, $\langle G \rangle \in ALL_{CFG}$ , then $\langle G, A \rangle \in \text{NECESSARY}_{CFG}$

Hence, *R* reduces $ALL_{CFG}$ to $\text{NECESSARY}_{CFG}$

Comments (6)

## Problem

Say that a CFG is *minimal* if none of its rules can be removed without changing the language generated. Let $MIN_{CFG}$ =

$$\{\langle G\rangle \mid G \text{ is a minimal CFG}\}.$$

**a.** Show that $MIN_{CFG}$ is T-recognizable.

**b.** Show that $MIN_{CFG}$ is undecidable.

## Step-by-step solution

**Step 1** of 6

a)

A minimal context-free grammar $MIN_{CFG}$ is one in which no rule can be modified without changing the language generated. Now, it can Proved that $MIN_{CFG}$ is T-recognizable.

• The grammar $MIN_{CFG}$ can be converted to an equivalent grammar in Chomsky normal form.

• The generated CFG can be simulated by a Turing machine because for a string of length $k$ there will be a finite number of derivations with $2k-1$ steps.

• This Turing machine $U$ will accept the strings that are in the language.

Comment

**Step 2** of 6

The Turing machine is:

$U =$"On input $\langle G, w\rangle$, where $G$ is a minimal CFG and $w$ is a string:

1. Create an equivalent grammar $H$ in Chomsky normal form from $G$.

2. Let $n = |w|$.

3. If $n = 0$, then check all the derivations with one step in $H$, else check all the derivations with $2n-1$ steps.

4. *Accept* if any of the derivations accept $w$, else *reject*.

Comments (1)

**Step 3** of 6

Therefore, **a minimal context-free grammar is T-recognizable.**

Comment

**Step 4** of 6

b)

Consider the $MIN_{CFG}$. Now, it can be shown that $MIN_{CFG}$ is un-decidable. It can be proved by contradiction, that is, by taking an assumption that $MIN_{CFG}$ is decidable. Assume the opposite, which is the grammar $MIN_{CFG}$ is decidable.

Comment

**Step 5** of 6

It is known that if a language is decidable/ undecidable if and only if the complement of this language is decidable/ undecidable. Now consider $ALL_{CFG}$, which is already known as undecidable.

On input $\langle G \rangle$:

• Construction of $G'$ takes place by adding to $G$ a new terminal, together with productions: $G'S \to A, A \to \varepsilon, and\ A \to aA\ for\ each\ a \in \Sigma$

• Output $\langle G', A \rangle$.

Comment

**Step 6** of 6

The construction of the grammar $G'$ is takes place by $f$ in such a way that it always show $L(G') = \Sigma^*$. Thus, if $L(G) = \Sigma^*$ then it is not necessary that $A$ is for $G'$.

• The reason behind it is that every string $l = \Sigma^*$ can already be derived from $G$, which shows a contradiction.

Hence, from the above explanation it can be said that," $MIN_{CFG}$ **is undecidable**".

Comment

## Problem

Give an example in the spirit of the recursion theorem of a program in a real programming language (or a reasonable approximation thereof ) that prints itself out.

## Step-by-step solution

**Step 1** of 2

**Recursion theorem:** Let T be a Turing machine that computes a function $t:\Sigma^*\times\Sigma^*\to\Sigma^*$. There is a Turing machine R that computes a function $r:\Sigma^*\to\Sigma^*$, where for every $w$,

$$r(w)=t\left(\langle R\rangle,w\right)$$

• The recursion theorem produces a new machine *R,* which operates exactly as *T* does.

• It has connection to the theory of self- reproducing system.

*SELF* = "On any input:

1. Obtain, via the recursion theorem, own description $\langle SELF\rangle$.

2. Print $\langle SELF\rangle$."

Comment

**Step 2** of 2

• In the real programming Language, LISP plays the same role as a recursion theorem.

• The following program in LISP is an example in the spirit of recursion theorem.

```
(define self (lambda (x)
  (list x (list (quote quote) x))
)
(quote(lambda (x)
(list x (quote quote) x)
)
)
```

(quote x) represents in Lisp as , (lambda(x) (list x(quote quote) x)) is printing of word x is initialization. Whole program represent self printing.

Comment

## Problem

Show that any infinite subset of $MIN_{TM}$ is not Turing-recognizable.

## Step-by-step solution

**Step 1** of 3

Definition of MIN $_{TM}$ =

If *M* is a Turing machine, then we say that the length of the description $\langle M \rangle$ of *M* is the number of symbols in the string description *M*.

Say that *M* is minimal if there is no Turing machine equivalent to *M* that has a shorter description. $\boxed{\mathrm{MIN}_{\mathrm{TM}} = \{\langle M \rangle \mid M \text{ is a minimal TM}\}}$

Comment

**Step 2** of 3

Now we have to prove that any infinite subset of $\mathrm{MIN}_{\mathrm{TM}}$ is not Turing – recognizable.

We will prove by taking contradiction.

We assume that there exists *A*, an infinite subset of $\mathrm{MIN}_{\mathrm{TM}}$, such that *A* is Turing – recognizable.

We know that

"A language is Turing – recognizable if and only if some enumerator enumerates it".

So, Let *E* be the enumerator that enumerates *A* .

By using this *E*, we construct another *TM* (Turing – machine) *N* as follows.

$N =$"On input *w*:

1. from recursion theorem. Own description $\langle c \rangle$ is obtained

2. Run the Enumerator *E* until a machine *P* is obtained with a longer

description than that of *N*.

3. Simulate *P* on input *w*".

Comment

**Step 3** of 3

As we know that $\mathrm{MIN}_{\mathrm{TM}}$ is infinite *A* is infinite subset of $\mathrm{MIN}_{\mathrm{TM}}$.

1. When *A* is infinite, *E*'s list must contain a *TM* with longer description than $N$'s description. So obviously *N* terminates with some *TM* P which is longer than *N*. Then *N* simulates *P* and so is equivalent to *P*.

2. It also notify that N is shorter than *P*. So *P* cannot be minimal. But *P* appears on the list that is produced by *E*.

3. *E's* list must contain a *TM* with longer description than *N*'s description.

From above three conditions we have a contradiction.

Thus our assumption that *A* is Turing – recognizable is wrong.

Therefore *A* is not Turing – recognizable.

Thus an infinite subset of $\mathrm{MIN}_{\mathrm{TM}}$ is not Turing – recognizable.

Comment

## Problem

Show that if A $\leq_T$ B and B $\leq_T$ C, then A $\leq_T$ C.

## Step-by-step solution

**Step 1** of 1

Definition of $A \leq_T B$:

Language *A* is Turing reducible to Language *B*, written as $A \leq_T B$, if *A* is decidable relative to *B*. That is the oracle for language *B* decides Language *A*.

Given that

- $A \leq_T B$

That is, Let the oracle $M_1^B$ for Language *B* decides the Language *A*. and

- $B \leq_T C$

That is, Let the oracle $M_2^C$ for Language *C* decides the Language *B*.

We have to prove that

- $A \leq_T C$

That is, there exists an oracle $M_3^C$ for Language *C* which decides the Language *A*.

That means, machine $M_3$ have to simulate machine $M_1$.

We will explain this simulation in detail as follows

• Let $M_1$ queries an oracle about some String *x*.

• $M_3$ does not have an oracle for *B*. $M_3$

• So $M_3$ does not perform the test whether $x \in B$ or not directly.

• Thus $M_3$ first simulates $M_2$ on input *x* and get the result.

• Then $M_3$provides that answers to $M_1$.

• But the queries of machine $M_2$ are directly answered by $M_3$. Because $M_3$ and $M_2$ use same oracle *C*.

In this way the oracle for Language *C* decides Language *A*. That is $A \leq_T C$.

Thus If $A \leq_T B$ and $B \leq_T C$ then $A \leq_T C$.

Comment

## Problem

Let

$$A_{\mathsf{TM}}' = \{\langle M, w\rangle \mid M \text{ is an oracle TM and } M^{A_{\mathsf{TM}}} \text{ accepts } w\}.$$

is undecidable relative to $A_{TM}$.

## Step-by-step solution

**Step 1** of 1

Given that

$$A_{TM}' = \{\langle M, w\rangle \mid M \text{ is an oracle TM (turing machine) and } M^{A_{TM}} \text{ accepts } w\}$$

We have to show that $A_{TM}'$ is undecidable relative to $A_{TM}$.

Take a contradiction of $A_{TM}'$ is decidable relative to $A_{TM}$.

Hence there exists an oracle *TM T* with oracle access to $A_{TM}$ which decides $A_{TM}'$.

Now we construct another oracle *TM N* as follows:

*N="on input*

1. Run $T^{A_{TM}}$ on input >

2. If *T* accepts, *reject*.

3. Else if *T* rejects, *accept*".

• So $N^{A_{TM}}$ accepts if and only if *M* rejects.

• When the input of *N* is <*N*>, we have $N^{A_{TM}}$ accepts <*N*> if and only if *N* rejects <*N*>.

This is a contradiction to our hypothesis, that $A_{TM}'$ is decidable relative to $A_{TM}$ is wrong. Hence, $A_{TM}'$ is un-decidable relative to $A_{TM}$.

Comment

## Problem

Is the statement $\exists x \, \forall y \, [\, x+y=y \,]$

## Step-by-step solution

**Step 1** of 1

Theory of *N* :-

If *N* is a model, theory of *N*, written $Th(N)$, be the collection of true sentences in the language of that model.

• Given sentence is $\phi_1 = \exists x \forall y [x+y=y]$ and the given theory of model is $Th(N,+)$.

The statement $\phi_1$ is true in model $(N,+)$.Because for $N=\{0,1,2,3...\}$ and for $x=0$ the statement $0+y=y$ is true.

So the statement $\exists x \forall y [x+y=y]$ is a member of $Th(N,+)$

• Given statement is $\phi_2 = \exists x \forall y [x+y=x]$

This statement $\phi_2$ is false in the model $(N,+)$

Because for any *x* the statement $x+y=x$ is false, for all $y$ (except if $y=0$).

So the statement $\phi_2$ is false in the model $(N,+)$

Hence the statement $\exists x \forall y [x+y=x]$ is not a member of $Th(N,+)$

Comment

## Problem

Describe two different Turing machines, M and N, where M outputs $\langle N \rangle$, when started on any input.

## Step-by-step solution

**Step 1** of 2

Given: There are two different Turing machines *M* and *N.*

We have to prove that, it started with any input *w*, *M* outputs $\langle N \rangle$ and *N* outputs $\langle M \rangle$

Here we need to know the recursion theorem.

**Recursion theorem:** Let T be a Turing machine that computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. There is a Turing machine R that computes a function $r : \Sigma^* \to \Sigma^*$, where for every $w$,

$r(w) = t(\langle R \rangle, w)$.

To make a Turing machine that can get its own description and computes with it. Machine T in the statement receives the description of machine as extra input. The recursion theorem produces a new machine R, which operates exactly T, does but R's description filled automatically.

Comment

**Step 2** of 2

By using Recursion theorem, remember that $r(w)$ is a Turing machine that prints *w* on its tape, and halts.

The Turing machine *M* is as follows:

$M =$" on input *w*:

1. No need to note the input

2. Obtain $\langle M \rangle$ by using the recursion theorem.

3. Compute $r(\langle M \rangle)$ where $r$ the computable function is write

$r(\langle M \rangle)$ on the tape.

4. Halt."

• When *M* runs, it writes $N = \langle r(\langle M \rangle) \rangle$ in its tape.

• When *N* runs, it writes $\langle M \rangle$ in its tape automatically.

$\langle N \rangle$ is a member of the turing machine $M$
$\langle M \rangle$ is a member of the turing machine $N$

Therefore *M* outputs $\langle N \rangle$ and *N* outputs $\langle M \rangle$.

Comment

## Problem

In the fixed-point version of the recursion theorem (Theorem 6.8), let the transformation t be a function that interchanges the states $q_{accept}$ and $q_{reject}$ in Turing machine descriptions. Give an example of a fixed point for t.

**THEOREM 6.8**

Let $t\colon \Sigma^* \longrightarrow \Sigma^*$ be a computable function. Then there is a Turing machine $F$ for which $t(\langle F\rangle)$ describes a Turing machine equivalent to $F$. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.

In this theorem, $t$ plays the role of the transformation, and $F$ is the fixed point.

## Step-by-step solution

**Step 1** of 4

Fixed point of a function is value that is not changed by application of the function. We need to consider functions are computable transformations of Turing machine descriptions. Turing machine behavior is unchanged by transformations that theorem called fixed-point version of recursion theorem.

Fixed – point version of the recursion theorem:- Let $t:\Sigma^* \to \Sigma^*$ be a computable function then there is a Turing machine *F* for which $t(\langle F\rangle)$ describes a Turing machine equivalent to *F*. If a string is not proper Turing machine encoding, then it describes Turing machine rejects immediately.

$t$: is a role of transformation

$F$: fixed point

Comment

**Step 2** of 4

Given that,

The transformation *t* is a function that interchanges the states $q_{accept}$ and $q_{reject}$ in Turing machine descriptions.

Now we have to define the fixed point for *t*.

Comment

**Step 3** of 4

Let $\langle M\rangle$ be a fixed point machine.

(i) *M* Halts on input *w*

• If *M* halts on any input *w*, then clearly *M* cannot be a fixed point since $q_{accept}$ and $q_{reject}$ are replacing.

• The Language recognized by $\langle M\rangle$ and the Language recognized by $t\langle M\rangle$ must differ in *w*.

• So *M* does not halt on any input, instead of that it rejects then only *M* accepts as fixed point.

(ii)

• Suppose *M* loops infinitely on all inputs.

• Then $(t\langle M\rangle)$ must also loop infinitely, because it is the same machine but with accept and reject states swapped.

• Therefore *M* is a fixed – point.

Hence any machine which loops infinitely on all inputs is a fixed – point since it rejects.

Comment

**Step 4** of 4

Example: A machine which can never leave the start state is an example of fixed point.

Start with $n$-state Turing machine start with blank tape but the productivity is zero.

Machine $M$ on blank input

- Get description for $\langle M \rangle$ from recursion theorem for $n$ states of $\langle M \rangle$
- Compute Productivity $p(n)$
- Add 1 to obtain $p(n) + 1$
- Once end of end of tape print $p(n) + 1$
- But $M$ is an $n$-state Turing machine that outputs more than the productivity function. It is a contradiction.

Comment

## Problem

Show that $EQ_{\mathrm{TM}} \not\leq_{\mathrm{m}} \overline{EQ_{\mathrm{TM}}}$.

## Step-by-step solution

**Step 1** of 3

**TM equality:**

The TM equality is represented as follows:

$$EQ_{TM} = \{(\langle M\rangle, \langle N\rangle) \text{ where } M \text{ and } N \text{ are Turing machines and } L(M) = L(N)\}.$$

$$EQ'_{TM} = \{(\langle M'\rangle, \langle N'\rangle) \text{ where } M' \text{ and } N' \text{ are Turing machines and } L(M') = L(N')\}$$

$EQ_{TM} \not\leq_m \overline{EQ_{TM}}$ means that $EQ_{TM}$ is not mapping reducible to $\overline{EQ_{TM}}$. This means that $EQ_{TM}$ is not mapping reducible to its complement.

Comment

**Step 2** of 3

**Proof:**

In order prove that $EQ_{TM} \not\leq_m \overline{EQ_{TM}}$, first prove that $EQ_{TM}$ is not Turing-recognizable.

According to Theorem 5.28 and Corollary 5.29, $A \leq_m B$ only if both $A$ and $B$ are Turing recognizable or not Turing recognizable.

$\overline{EQ_{TM}}$ is complement of $EQ_{TM}$. So, if $EQ_{TM}$ is not Turing-recognizable then, $\overline{EQ_{TM}}$ is Turing-recognizable and vice-versa. This, result in not mapping reducibility between $EQ_{TM}$ and $\overline{EQ_{TM}}$.

Comment

**Step 3** of 3

**Example:**

Assume $A_{TM}$ is a Turing machine and is mapping is mapping reducible to $\overline{EQ_{TM}}$ that is $A_{TM} \leq_m \overline{EQ_{TM}}$.

The function $f_2 : A_{TM} \to \overline{EQ_{TM}}$ is defined as follows:

$$\begin{aligned}
&f_2 : On\ input\ \langle M, w\rangle \\
&\quad Construct\ machine\ M_3 : on\ any\ input,\ reject. \\
&\quad Construct\ machine\ M_4 : on\ any\ input\ x, run\ M\ on\ w. \\
&\qquad\qquad If\ it\ accepts,\ accept\ x. \\
&\quad Output\ \langle M_3, M_4\rangle
\end{aligned}$$

**Explanation:**

- The machine $M_1$ accepts nothing.
- If $M$ accepts $w$, then $M_2$ accepts everything. Otherwise it accepts nothing.
- So, $\langle M, w\rangle \in A_{TM} \Leftrightarrow \langle M_3, M_4\rangle \in \overline{EQ_{TM}}$ and $f_2$ is clearly computable. Thus, it is a reduction from $A_{TM}$ to $\overline{EQ_{TM}}$.
- So, $EQ_{TM}$ is not Turing-recognizable.

Thus, if $EQ_{TM}$ is not Turing-recognizable and $\overline{EQ_{TM}}$ is Turing-recognizable then $EQ_{TM}$ is not mapping reducible to $\overline{EQ_{TM}}$. That is, $EQ_{TM} \not\leq_m \overline{EQ_{TM}}$.

## Problem

Use the recursion theorem to give an alternative proof of Rice's theorem in Problem 5.28.

Problem 5.28

**Rice's theorem**. Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language—whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$. Here, $M_1$ and $M_2$ are any TMs. Prove that P is an undecidable language.

## Step-by-step solution

**Step 1** of 2

Rice's theorem: Let *P* be any nontrivial property of the language of a Turing machine.

In more formal terms, Let *P* be a Language consisting of Turing machine descriptions where *P* fulfills two conditions.

• First, *P* is nontrivial – It contains some, but not all, *TM* descriptions.

• Second, *P* is a property of the *TM's* language- whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$. Here $M_1$ and $M_2$ are any $TMs$.

Then we have to prove that *P* is an undecidable Language.

Assume that *P* is a decidable Language satisfying the Properties of Rice theorem.

Comment

**Step 2** of 2

Let the Turing machine *T* decides the language *P*. According to the first condition of Rice theorem, *P* contains some, but not all, *TM* descriptions.

From above properties we assume $TM_1 \& TM_2$ be the Turing machines and $\langle TM_1 \rangle \in P$ and $\langle TM_2 \rangle \notin P$.

Now construct the Turing machine $TM_3$ using $TM_1 \& TM_2$.

$TM_3$"on input *w*:

1. By using recursion theorem, construct our own description for $\langle TM_3 \rangle$

2. Run *T* on Turing machine $\langle TM_3 \rangle$

3. If *T* accepts $\langle TM_3 \rangle$, simulate $\langle TM_2 \rangle$ on *w*.

If *T* rejects $\langle TM_3 \rangle$, simulate $\langle TM_1 \rangle$ on *w*.

• Suppose $\langle TM_3 \rangle \in P$

Then *T* accepts $\langle TM_3 \rangle$ and $L(TM_3) = L(TM_2)$

According to second condition of Rice's theorem $\langle TM_2 \rangle \in P$ but our condition is $\langle TM_2 \rangle \notin P$. It is a contradiction here.

• So, we will get similar contradiction if $\langle TM_3 \rangle \notin P$.

As a result of that our assumption is wrong for *P* is a decidable language. Hence every property satisfying the conditions of Rice's theorem is undecidable.

Comment

## Problem

Give a model of the sentence

$$\begin{aligned}\phi_{\text{eq}} \;=\; & \forall x\,\left[\,R_1(x,x)\,\right] \\ & \wedge \forall x,y\left[\,R_1(x,y) \leftrightarrow R_1(y,x)\,\right] \\ & \wedge \forall x,y,z\left[\,(R_1(x,y) \wedge R_1(y,z)) \rightarrow R_1(x,z)\,\right].\end{aligned}$$

## Step-by-step solution

**Step 1** of 5

Given sentence is

$$\begin{aligned}\phi_{eq} = & \forall x\left[R_1(x,x)\right] \\ & \wedge \forall x,y\left[R_1(x,y) \leftrightarrow R_1(y,x)\right] \\ & \wedge \forall x,y,z\left[\left(R_1(x,y)\right) \wedge R_1(y,z) \rightarrow R_1(x,z)\right]\end{aligned}$$

$\phi_{eq}$ gives three conditions of equivalence relations i.e., Reflexive relation, Symmetric relation and Transitive relations.

Comment

**Step 2** of 5

Let $R_{11}$ and $R_{12}$ are two equivalence relations on some set, definition of $R_1$ by

$$\forall x,y\left[R_1(x,y) \equiv R_{11}(x,y) \wedge R_{12}(x,y)\right]$$

Where *x, y* are elements from set.

**Reflexive relation:** $\forall x\left[R_1(x,x)\right]$

$$\begin{aligned}& \equiv \{\text{definition of } R_1\} \\ & R_{11}(x,x) \wedge R_{12}(x,x)\end{aligned}$$

$$\begin{aligned}& \equiv \{R_{11}, \text{being an equivalence relation, is reflexive}\} \\ & \text{similarly } R_{12}\end{aligned}$$

$$\begin{aligned}& \equiv \{true\} \wedge \{true\} \\ & \equiv true\end{aligned}$$

Comment

**Step 3** of 5

**Symmetric relation:** $\forall x,y\left[R_1(x,y) \leftrightarrow R_1(y,x)\right]$

$$\begin{aligned}& \equiv \{\text{definition of } R_1\} \\ & R_{11}(x,y) \wedge R_{12}(x,y)\end{aligned}$$

$$\begin{aligned}& \equiv \{R_{11}, \text{being an equivalence relation, is symmetric}\} \\ & \text{similarly } R_{12}\end{aligned}$$

$\equiv$ {definition of $R_1$}

$R_1(y,x)$

Comment

**Step 4** of 5

**Transitive relation:** $\forall x,y,z\left[\left(R_1(x,y)\right)\wedge R_1(y,z)\rightarrow R_1(x,z)\right]$

$\equiv$ {definition of $R_1$}

$\left(R_{11}(x,y)\wedge R_{12}(x,y)\right)\wedge\left(R_{11}(y,z)\wedge R_{12}(y,z)\right)$

$\equiv$ {rearranging the conjucts}

$\left(R_{11}(x,y)\wedge R_{11}(y,z)\right)\wedge\left(R_{12}(x,y)\wedge R_{12}(y,z)\right)$

$\equiv$ {$R_{11}$, being an equivalence relation, is transitive}

similarly $R_{12}$

$\equiv$ {definition of $R_1$}

$R_1(x,z)$

Comment

**Step 5** of 5

A model $(U,R_1)$, where *U* is universe and $R_1$ is equivalence relation over *U,* is a model of $\phi_{eq}$.

Comment

## Problem

Let $\phi_{\text{eq}}$ be defined as in Problem 6.10. Give a model of the sentence

$$\begin{aligned}\phi_{\text{lt}} \;=\; & \phi_{\text{eq}} \\ & \wedge \forall x,y \left[ R_1(x,y) \rightarrow \neg R_2(x,y) \right] \\ & \wedge \forall x,y \left[ \neg R_1(x,y) \rightarrow (R_2(x,y) \oplus R_2(y,x)) \right] \\ & \wedge \forall x,y,z \left[ (R_2(x,y) \wedge R_2(y,z)) \rightarrow R_2(x,z) \right] \\ & \wedge \forall x \, \exists y \left[ R_2(x,y) \right].\end{aligned}$$

Give a model of the sentence

$$\begin{aligned}\phi_{\text{eq}} \;=\; & \forall x \left[ R_1(x,x) \right] \\ & \wedge \forall x,y \left[ R_1(x,y) \leftrightarrow R_1(y,x) \right] \\ & \wedge \forall x,y,z \left[ (R_1(x,y) \wedge R_1(y,z)) \rightarrow R_1(x,z) \right].\end{aligned}$$

## Step-by-step solution

**Step 1** of 2

Consider the following sentence that is provided in the problem 6.10 in the textbook,

$$\begin{aligned}\phi_{eq} = & \forall x \; [R_1(x,x)] \\ & \wedge \forall x,y \; [R_1(x,y) \leftrightarrow R_1(y,x)] \\ & \wedge \forall x,y,z \; [(R_1(x,y) \wedge R_1(y,z)) \rightarrow R_1(x,z)]\end{aligned}$$

The above statement explains about the conditions of the equivalence relation. A model $(A, R_1)$, where *A* is any universe and $R_1$ is the equivalence relation over the elements of *A*. The line 1 describes that for all *x*, *x* is equal to itself. The line 2 describes that for all *x* and *y*, if and only if *x* is equal to *y* then *y* is equal to *x*. The line 3 describes that for all *x, y,* and *z*, if *x* is equal to *y* and *y* is equal to *z* then *x* is equal to *z*.

Comment

**Step 2** of 2

Consider the following sentence,

$$\begin{aligned}\phi_{lt} = & \phi_{eq} \\ & \wedge \forall x,y \; [R_1(x,y) \rightarrow \neg R_2(x,y)] \\ & \wedge \forall x,y \; [\neg R_1(x,y) \rightarrow (R_2(x,y) \oplus R_2(y,x))] \\ & \wedge \forall x,y,z \; [(R_2(x,y) \wedge R_2(y,z)) \rightarrow R_2(x,z)] \\ & \wedge \forall x \exists y \; [R_2(x,y)]\end{aligned}$$

The above statement explains about the conditions of the equivalence relation and less than relation. A model $(A, R_1, R_2)$, where *A* is any universe, $R_1$ is the equivalence relation over the elements of *A* and $R_2$ is the less than relation over *A*.

• The line 1 describes the conditions of the equivalence relation.

• The line 2 describes that for all $x$ and $y$, if $x$ is equal to $y$ then it is tends to complement of less than relation. This means if $x$ is equal to $y$ then it can be said that $x$ is not less than $y$.

• The line 3 describes that for all $x$ and $y$, if $x$ is not equal to $y$ then either $x$ less than $y$ or $y$ less than $x$ is tru

• The line 4 describes that for all $x$, $y$ and $z$, if $x$ is less than $y$ and $y$ is less than $z$ then $x$ is less than $z$.

• The line 5 describes that for all $x$ there exist $y$, $x$ is less than $y$.

Comment

## Problem

Let $(\mathcal{N}, <)$ is decidable.

## Step-by-step solution

**Step 1** of 2

Consider theorem 6.12 given in the textbook where it is proved that Th(◊, +) is decidable where + refers to the relation. As decidability has been proved for Th(◊,+), reduce Th(◊,+) to Th(◊,<).

The decidability of Th(◊, <) can be proved by converting sentence $S_1$ of language Th(◊,<) into $S_2$ sentence preserving all the truth or falsity related to models.

Comment

**Step 2** of 2

Now, replace the occurrences of $S_1$ in $S_2$ where $\wedge$ and $\vee$ with formula:

$$\exists k[(i+k=j)\wedge(k+k\neq k)]$$

*k* refers to new variable which will be different for every case.

$S_2$ is equivalent to $S_1$ because the value of *i* is less than the value of *j*. So, the value of *j* can be obtained by adding non-zero value in *i*. "i is less than *j*" that means '*j*' can be obtained by adding non zero value to '*i*'.

To prove the decidability of Th(◊,+) $S_2$ is supposed to be put in prenex-normal form.

For bringing existential qualifiers to front of the sentence, quantifiers are supposed to pass through Boolean operations that are being appeared in the sentence.

When the quantifiers are brought, the operations ^ and V will not be changed. When a null is brought it changes to and vice-versa. So, $\neg\exists k\varphi$ becomes $\forall k\neg\varphi$ and $\neg\forall k\varphi$ becomes $\exists k\neg\varphi$.

So, Th(◊,<) is decidable.

Comment

## Problem

For each

$$m > 1 \text{ let } \mathcal{Z}_m = \{0, 1, 2, \ldots, m-1\}, \text{ and let } \mathcal{F}_m = (\mathcal{Z}_m, +, \times)$$

is decidable.

## Step-by-step solution

**Step 1** of 1

Given that

- $z_m = \{0,1,2,...,m-1\}$ for $m > 1$ is a universe.
- $F_m = (z_m, +, \times)$ be the model whose universe is $z_m$.
- $+$ and $\times$ are relations over modulo *m*.

Now we have to show that $Th(F_m)$ is decidable.

Clearly $z_m$ is finite.

So we can simply enumerate all the possible values into the formula and see if the formula is true.

If it is true then it is belong to $Th(F_m)$

If it is false then it does not belong to $Th(F_m)$

In particular, the following recursive procedure would work:

- Let $\exists x_i$ such that $\phi_i(x_1, x_2, ...x_i)$ be the formula
- Substitute $x_i = 0,1,...,m-1$ into the formula.
- If the formula $\phi_i(x_1, x_2, ...x_i)$ is true for any value of $x_i$ then the original formula is true
- That means, if we have $\forall x_i$ instead of $\exists x_i$ then the original formula will be true.

$\rightarrow$ A theory of a model is said to be decidable if we say that the formulae which are true are belong to that model and which are false that does not belong to that model.

So by the above recursive procedure we say that $Th(F_m)$ is decidable.

Comment

## Problem

Show that for any two languages A and B, a language J exists, where A $\leq_T$ J and B $\leq_T$ J.

## Step-by-step solution

**Step 1** of 1

For any two languages A and B, a language J exists, where $A \leq_T J$ and $B \leq_T J$. It can be proved as follows:

Two languages A and B are given and one more language J exists to find Turing reducibility between A and J and between B and J. it can be proved by using the concept of mapping reducibility.

$A \leq_T J$ means that A is Turing reducible to language J. $B \leq_T J$ means that B is Turing reducible to J. Now, use the concept of decidability. Here J is an intermediate language that is being used to prove the Turing reducibility of A and B. So prove first of all decidability of J must be proved.

Decidability of J can be proved by the given relation $B \leq_T J$ and $A \leq_T J$ if A and B are decidable then J will also be decidable. Replace the oracle Turing machine that decides A by an ordinary Turing Machine that decides A

Turing reducibility is the generalized form of mapping reducibility. Mapping reducibility holds true only if decidability holds true because the mapping reducibility may be used to give an oracle Turing Machine that decide Turing reducibility of A and B on the Basis of J.

Comments (8)

## Problem

Show that for any language A, a language B exists, where $A \leq_{\mathrm{T}} B$ and $B \not\leq_{\mathrm{T}} A.$

## Step-by-step solution

**Step 1** of 3

Given that

*A* and *B* are two Languages.

We have to show that

$A \leq_T B$ and $B \not\leq_T A$.

That means

• *A* is Turing reducible to *B*.

But *B* is not Turing reducible to *A*.

Let $A = E_{TM}$ (Empty Turing machine)

$= \{\langle M \rangle | \text{ M is a TM and L(M)} = \phi\}$

And $B = A_{TM}$ (Oracle Turing machine)

$= \{\langle M, w \rangle, \text{ machine } M \text{ accepts } w\}$

Comment

**Step 2** of 3

(i) $A \leq_T B$ :

To show $A \leq_T B$ , we have to show that $E_{TM} \leq_T A_{TM}$ .

We know that $E_{TM}$ is decidable relative to $A_{TM}$ .

Since $E_{TM} \leq_T A_{TM}$ , then *A* is decidable relative to *B* then $A \leq_T B$ .

Comments (1)

**Step 3** of 3

(ii) $B \not\leq_T A$

To shown $B \not\leq_T A$ we have to show that

$A_{TM} \not\leq_T E_{TM}$ .

Let us assume the contradiction $A_{TM} \leq_T E_{TM}$ .

By mapping reducibility rules for any two languages $x \leq_m y \Leftrightarrow \overline{x} \leq_m \overline{y}$

Language x is mapping reducible to language y, written $x \leq_m y$ , if there is a computable function $f : \Sigma^* \to \Sigma^*$ , where for every *w, f* called reduction of *x* to *y*.

$w \in x \Leftrightarrow f(w) \in y$

According to given mapping reducibility we have to derive $A_{TM} \leq_T E_{TM} \Leftrightarrow \overline{A_{TM}} \leq_m \overline{E}_{TM}$

However $\overline{E}_{TM}$ is Turing recognizable, but $\overline{A}_{TM}$ is not Turing recognizable

Accor ¨ ıg mapping reducibility rules

$x \leq_m y$ and *y* is Turing – recognizable then *x* must be Turing recognizable.

Therefore this gives a contradiction.

Therefore our assumption that $A_{TM} \leq_T E_{TM}$ is wrong.

Hence $A_{TM} \not\leq_T E_{TM}$ i.e., $B \not\leq_T A$.

From (i) and (ii) we have showed that there exist two languages *A* and *B* such that

$A \leq_T B$ But $B \not\leq_T A$.

Comments (1)

## Problem

Prove that there exist two languages A and B that are Turing-incomparable—that is, where $A \not\leq_{\mathrm{T}} B$ and $B \not\leq_{\mathrm{T}} A$.

## Step-by-step solution

**Step 1** of 1

**Given:**

In this problem two languages $A$ and $B$ are present.

**Proof:**

Two languages $A$ and $B$ are Turing incomparable if the language is not Turing comparable with language $B$ and it's vice versa.

Languages Incomparability can be proved as follows:

• In the Previous question there was one more language $J$ was given for proving Turing reducibility of 2 languages $A$ and $B$.

• If it comes to previous situation there was one more language $J$ exists and it is being used to decide the Turing reducibility of two languages $A$ and $B$.

• But if it comes to discuss about current situation then no other language or can say oracle support with Turing machine or language is given to detect Turing Comparability.

• Here no Turing machine is given that will work as an oracle Turing machine to find decidability and Turing reducibility.

• So none language from $A$ and $B$ can be replaced by any intermediate language and further no oracle Turing machine can be produced.

• For producing any Turing machine at least one intermediate language is required.

• So both languages can be compared through intermediate languages and Turing reducibility can be found.

• Oracle Turing machine set is countable that is why it is required to find the Turing reducibility and on the other hand set of all languages are uncountable.

• Most important thing is still missing that Turing reducibility cannot be established until decidability is proved as Turing reducibility is generalized concept of decidability.

• If discuss about further concept that how to find Turing comparability then machine must be added up with oracle support.

**Conclusion:**

No solution or method is defined here to prove the decidability of 2 languages so languages $A$ and $B$ are not Turing reducible.

Hence it is clear that language $A$ and $B$ is Turing incomparable.

Comment

## Problem

Let A and B be two disjoint languages. Say that language C *separates* A and B if $A \subseteq C$ and $B \subseteq \overline{C}$. Describe two disjoint Turing-recognizable languages that aren't separable by any decidable language.

## Step-by-step solution

**Step 1** of 2

Let $A$ and $B$ are two disjoint languages. Consider that language $C$, that separates $A$ and $B$ if $A \subseteq CandB \subseteq \overline{C}$. It can be proved as follows:

• It is quite easy to understand this statement. Read the statement carefully things are pretty obvious all you need to understand is $A \subseteq CandB \subseteq \overline{C}$.

• Here, $A$ **is a set of** $C$ **and** $B$ **is a set of Complement of** $C$. So, it is quite obvious now $A$ and $B$ both are not related to each other anyhow.

• Here, no use of $C$ because even if we use $C$ it won't be able to prove decidability of $A$ on the basis of $B$ and Turing reducibility of $A$ on the basis of $B$.If it comes to languages those are fully different and belong to different sets then no separators are required. So, here it is worthless to use $C$ as separator.

Comment

**Step 2** of 2

Now, consider a Turing machine $T$ that will work as a decider for the language C that separates $A$ and $B$.**Consider both languages as Regular Expressions that will be decided by** $M_1$ **and** $M_2$**.**

1. $S = <M, w>$ Where $M$ is a Turing machine.

2. Now, run $< M_1, W >$ and $< M_2, W >$

3. If $M_1$ accepts then $M$ rejects and if $M_2$ accepts $M$ rejects.

• **Remember** $M$ **will always halt in each situation. Where C decides A or B. Now it is pretty easy to understand the situation**.

• Therefore, it can be said that only the first statement of question is enough to prove the concept" **No decidable languages can be used to separate two disjoint Turing recognizable languages**".

Comment

## Problem

Show that $\overline{EQ_{\text{TM}}}$ is recognizable by a Turing machine with an oracle for $A_{TM}$.

## Step-by-step solution

**Step 1** of 1

**Given:**

A language that is recognizable by an oracle Turing machine.

**Construction:**

Consider Turing Machine $T$ with an oracle for $A_{TM}$ and consider 2 more Turing machines $P$ and $Q$ that will work as decider for $T$ and then $T$ will work as a decider for $A_{TM}$ and further $A_{TM}$ will work as a Turing recognizable for $\overline{EQ_{TM}}$. Use the following approach:

$(P, w) \in A_{TM} \longleftrightarrow T(< M, w >) \in EQ_{TM}$

Above statement can be applied by using following algorithm:

$T$ = on input $\langle P, w \rangle$

Where $w$ is an input string and running on $P$

• Construct 2 Machines $P$ and $Q$

$P$ = On any input: accept.

$Q$ = On any input: run $P$ on $w$, if it accepts, accept

• Output $\langle P, Q \rangle$

**Conclusion:**

Here, it is quite obvious that $T$ is working as a decider for $A_{TM}$ and then $A_{TM}$ for $\overline{EQ_{TM}}$. This way $\overline{EQ_{TM}}$ is recognizable by a Turing Machine with an oracle for $A_{TM}$.

Comment

## Problem

In Corollary 4.18, we showed that the set of all languages is uncountable. Use this result to prove that languages exist that are not recognizable by an oracle Turing machine with an oracle for $A_{TM}$.

COROLLARY **4.18**

Some languages are not Turing-recognizable.

## Step-by-step solution

**Step 1** of 1

**Given:**

Group or set of all languages are not countable.

**Proof:**

From the corollary 4.18 of text book it is clear that the group of legal encodings of a Turing machine is countable with $A_{TM}$ accessed by oracle. Use same argument for proving that the set of all Turing machines is countable.

In other words it can be said that Turing machine with oracle is countable but on the other hands decidability, reducibility and recognizability cannot be proved without oracle support.

But the set of all languages are not countable. Any language $D$ can be mapped to binary vector corresponding of their characteristic vector. This binary vector is infinite in nature and composed of two sequences which are 0's and 1's. The characteristic vector is represented as $\chi_D$ .

So group of all languages cannot be placed in correspondence to group of all Turing machines.

**Conclusion:**

Sometimes some set of all languages cannot be put into a correspondence with the set of all Turing Machines with oracle support. As the set of all languages cannot be kept into set of all Turing machines, hence some languages are not recognizable by using Turing machine oracle access to $A_{TM}$ .

Comments (1)

## Problem

Recall the Post Correspondence Problem that we defined in Section 5.2 and its associated language *PCP*. Show that *PCP* is decidable relative to $A_{TM}$.

## Step-by-step solution

**Step 1** of 1

**Given:**

Here, $A_{TM}$ is with associated with oracle concept.

**Proof:**

PCP (Post Correspondence Problem) is decidable relative to $A_{TM}$. Here, Post Correspondence Problem can be considered as an example of un-decidability problem concerning with manipulation of strings to find a match. A match can be found, if String made by combining all symbols of upper side and string made by combing all symbols of lower side, both is same.

Consider a contrary that PCP is un-decidable relative to $A_{TM}$ but it is not associated with oracle concept.

The concept of oracle associated with $A_{TM}$ provides flexibility or countable property that is not provided by other Turing machines that are not associated with oracle.

Now, it is clear that first found a match to find PCP. It is considered that PCP is un-decidable relative to $A_{TM}$ but it is not associated with oracle concept. Here, it is quite obvious if terminology of Turing machine is not associated with concept of oracle then it will be un-decidable rather PCP (Post Correspondence Problem) is decidable relative to $A_{TM}$. Here $A_{TM}$ is with associated with oracle concept

**Conclusion:**

Hence, PCP is decidable with respect to $A_{TM}$.

Comment

## Problem

Show how to compute the descriptive complexity of strings K(x) with an oracle for $A_{TM}$.

## Step-by-step solution

**Step 1** of 2

Descriptive complexity of strings:-

If *x* be binary string, then the minimal description and descriptive complexity of *x*'s are $d(x)$ and $\mathrm{K}(x)$ respectively. Turing machine *M* and small string *w* we get minimal description is $\langle M, w\rangle$. From several of such shorter strings we select lexicographically among them then we can get descriptive complexity of such strings $\mathrm{K}(x) = |d(x)|$.

Comment

**Step 2** of 2

Now we have to show how to compute $\mathrm{K}(x)$ with an oracle for $A_{TM}$.

• For the given string *x*, start testing all the strings '*S*' up to the length $|x| + c$

Where $c =$ length of *TM* (Turing machine) that halts immediately upon starting.

• All the strings up to the length $|x| + c$ are potential description of *x*.

• If *S* is well formed as $\langle M, w\rangle$ from all binary strings in lexicographic order, then we simulate *M* with input *w* and see if it halts with *x* on the tape.

• Here we do not know whether *M* will halt on input *w* or not.

• An oracle for $A_{TM}$ can determine this.

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a } TM \text{ and } M \text{ accepts } w\}$$

• An oracle for $A_{TM}$ will take $\langle M, w\rangle$ as input and determine whether *M* accepts *w* or not.

• If *M* doesn't halt we move on to the next string *S*, and so on.

• After that we will find lexicographically first string *S* among them.

• In this way shortest string will be determined and it is represented as minimal description $d(x)$.

• From $d(x)$, we find $\mathrm{K}(x)$ as

$$\mathrm{K}(x) = |d(x)|$$

• By this procedure, we will compute $\mathrm{K}(x)$ with an oracle for $A_{TM}$.

Comment

## Problem

Use the result of Problem 6.21 to give a function f that is computable with an oracle for $A_{TM}$, where for each n, f(n) is an incompressible string of length n.

## Step-by-step solution

**Step 1** of 2

Compute descriptive complexity of strings $\mathrm{K}(x)$ with an oracle for $A_{TM}$.

• For the given string *x*, start testing all the strings '*S*' up to the length $|x|+c$

Where $c=$ length of *TM* (Turing machine) that halts immediately upon starting.

• All the strings up to the length $|x|+c$ are potential description of *x*.

• If *S* is well formed as $\langle M,w\rangle$ from all binary strings in lexicographic order, then we simulate *M* with input *w* and see if it halts with *x* on the tape.

• Here we do not know whether *M* will halt on input *w* or not.

• An oracle for $A_{TM}$ can determine this.

$$A_{TM}=\{\langle M,w\rangle \mid M \text{ is a } TM \text{ and } M \text{ accepts } w\}$$

• An oracle for $A_{TM}$ will take $\langle M,w\rangle$ as input and determine whether *M* accepts *w* or not.

• If *M* doesn't halt we move on to the next string *S*, and so on.

• After that we will find lexicographically first string *S* among them.

• In this way shortest string will be determined and it is represented as minimal description $d(x)$.

• From $d(x)$, we find $\mathrm{K}(x)$ as

$$\mathrm{K}(x)=|d(x)|$$

• By this procedure, we will calculate $\mathrm{K}(x)$ with an oracle for $A_{TM}$.

Comment

**Step 2** of 2

From the above procedure, $K(x)$ is the descriptive complexity of strings and computed with an oracle $A_{TM}$.

Now, we have to know the definition of incompressible strings.

<u>Incompressible strings:</u>

Let *x* be a string. If *x* doesn't have any description shorter than itself then *x* is incompressible.

Let *f* be the function that is computable with an oracle for $A_{TM}$, for each *n*, calculate *f*(*n*) i.e., incompressible length of string *n.*

Create an oracle $TM$ (Turing machine) that does the following :

1. For a given number *n*, begin

2. Enumerate all strings *x* of length *n.*

3. For each *x*, calculate $\mathrm{K}(x)$ by using oracle for $A_{TM}$.

4. As soon as we find $x=f(n)$ with $\mathrm{K}(x)>=|x|$ output *x*.

[ $K(x)>=|x|$ means getting a string which is greater than its description.]

5. Halt.

Comment

## Problem

Show that the function K(x) is not a computable function.

## Step-by-step solution

**Step 1** of 4

Definition of complexity strings $\mathrm{K}(x)$ :-

Comment

**Step 2** of 4

If *x* be binary string, then the minimal description and descriptive complexity of *x*'s are $d(x)$ and $\mathrm{K}(x)$ respectively. Turing machine *M* and small string *w* we get minimal description is $\langle M, w \rangle$. From several of such shorter strings we select lexicographically among them then we can get descriptive complexity of such strings $\mathrm{K}(x) = |d(x)|$.

Comment

**Step 3** of 4

Now we must prove that $\mathrm{K}(x)$ is not a computable function.

Let $y_n$ be the lexicographical first-string s that satisfies $n < K(y)$. Let M be the Turing machine such that if the input is n, the binary strings that are generated are $x_0, x_1, x_2, x_3 \cdots$.

The Turing machine M computes $\mathrm{K}(x_i)$ and if $\mathrm{K}(x) > n$, then the Turing Machine will generate $x_i$ as the output and the machine will halt.

If the machine does not halt, the machine will examine the next lexicographical string $x_{i+1}$.

The Turing machine M will come across a string x which satisfies $\mathrm{K}(x) > n$ because K is unbounded.

Comment

**Step 4** of 4

For input n, the output generated is $y_n$ but the length of the input n is $\log_2(n)$. So, $K(n) < \log_2(n) + c$, where c is the constant.

So, for all n, $n < K(y_n)$, so it can be said that $n < \log_2(n) + c$ $n < \log_2(n) + c$ but this is false, if n is large.

This is a contradiction because M cannot compute K(x) so, it is not a computable function.

Comment

## Problem

Show that the set of incompressible strings is undecidable.

## Step-by-step solution

**Step 1** of 2

Incompressible strings:

Let $w_i$ be a string. If $w_i$ doesn't have any description shorter than itself then $w_i$ is incompressible.

Now we have to show that set of incompressible strings is un-decidable.

Let *A* be the set of incompressible strings and assume the contradiction *A* is decidable.

We construct a machine *M* which enumerates *A*.

**Enumeration:** $f : A \to N$ such that $f(w1) = 1, f(w2) = 2, f(w3) = 3...$ where first, second, and third shortest strings are respectively *w*1, *w*2, & *w*3.

Since *A* reaches infinite there is a string $w_i \in A$.

Comment

**Step 2** of 2

Define a Turing machine *T* which computes $w_i$ incompressible string of length *n*

$T =$ " on input *n*

1. Returns the first string $w_i$ that *M* enumerates of length *n*.

2. If $K(\langle T, n\rangle) = c + \log(n)$. For any constant *c*

Then we find *n* such that

$|w_i| = n > c + \log(n)$

The string $w_i$ is shorter description on $\langle M', f(w_i)\rangle$. Where $M'$ is a machine, $f(w_i)$ is input and output as $w_i$.

Run machine $M'$ each string in lexicographic order from and output the same from *M*.

It contradicts that $w_i$ is compressible. Therefore our assumption that "*A* is decidable" is wrong. So for *A* set of incompressible strings *A* is un-decidable.

Comment

## Problem

Show that the set of incompressible strings contains no infinite subset that is Turing-recognizable.

## Step-by-step solution

**Step 1** of 2

Incompressible strings:

Let $w_i$ be a string. If $w_i$ doesn't have any description shorter than itself then $w_i$ is incompressible.

Now we have to show that set of incompressible strings contains no infinite subset that is Turing recognizable.

Let *A* be the set of incompressible strings and assume the contradiction *A* contains infinite subset and Turing recognizable.

We construct an enumeration function $f : N \to A$, *A* is recognized by machine *M.*

**Enumeration:** $f : N \to A$ such that $f(1) = w1, f(2) = w2, f(3) = w3...$ where first, second, and third enumerated strings are respectively *w*1, *w*2, *w*3, etc.

Since *A* reaches infinite there is a string $w_i \in A$.

Comment

**Step 2** of 2

Turing machine *N* which computes an incompressible string of length at least *n*.

$N =$" On input *n* (an integer in binary notation)

1. Call *M* to enumerate incompressible strings $w_i$.

2. If $|w_i| >= n$, output $w_i$ and halt"

Now the length of $\langle N \rangle n$ is

$$|\langle N \rangle n| = |\langle N \rangle| + \log(n)$$

$= \log(n) + c$ where *c* is a constant

Now $|w_i| = n$

$n > \log n + c$ for large value of *n.*

$$n > |\langle N \rangle n| \qquad \left[\text{As} |\langle N \rangle| n = \log n + c\right]$$

Therefore $|w_i| > n$

This contradicts the incompressibility of string $w_i$, therefore our assumption that *M* enumerates an infinite subset of incompressible strings $w_i$ is wrong.

Hence set of incompressible strings contains no infinite subset is Turing – recognizable.

Comment

## Problem

Show that for any c, some strings x and y exist, where $$\mathrm{K}(xy) > \mathrm{K}(x) + \mathrm{K}(y) + c.$$

## Step-by-step solution

**Step 1** of 1

**Given:**

Consider that *x* and *y* are strings whereas *c* is constant.

**Proof:**

Consider a Turing machine *TM* having input *w*. Assume that *x* is a binary string in *w*. The minimal description $d(x)$ is the shortest string in *w*. The descriptive complexity of *x* is

$$K(x) = |d(x)|$$

Consider a string *xy* that has been broken in 2 parts *x* and *y* and *c* can be considered as a default value that is used to match left hand side with right hand side.

Now use a Turing machine *T* that breaks its input *w* in description of 2 separate parts $d(x)$ and $d(y)$. The bits of first part and bits of second part are combined together or concatenate.

Once both of the part are obtained than both parts can be run to obtain the string *x* and *y* and can perform concatenate operation to achieve desired output of *xy*.

The length of description of part *xy* is clearly greater than individual parts description of *x* and *y* as some extra part *c* can be seen in the statement. The constant *c* can be termed as a constant for balancing between *xy* and individual parts.

**Conclusion:**

Hence, the Description complexity of *xy* is greater than sum of description of *x, y* and constant *c*.

$$K(xy) = K(x) + K(y) + c$$

Comments (2)

## Problem

Let $S = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \{\langle M \rangle\}\}.$ is Turing-recognizable.

## Step-by-step solution

**Step 1** of 1

$S$ and $\overline{S}$ are not recognizable can be proof by following method:

**Proof:**

This can be proved by reduction that is by reducing $\overline{A_{TM}}$ to $S$ and $\overline{A_{TM}}$ to $\overline{S}$. At first it is necessary to show that $A_{TM} \leq_m S$.

Consider a function *f* having input $\langle M, w \rangle$ and output $M'$ which works as follow on input *x*.

1. When $x == \langle M' \rangle$

**ACCEPT**

2. When $x == 0$ then execute $M$ on input *w*.

If *w* is accept the **ACCEPT** else **REJECT**

3. **REJECT.**

It is a many-one reduction from $\overline{A_{TM}}$ to $S$. When $\langle M, w \rangle \in A_{TM}$ then it shows that *w* is accepted by *M*. According to $M'$ definition it can be states that $L(M) = \{\langle M' \rangle, 0\} \neq \{\langle M' \rangle\}$. When $\langle M, w \rangle \notin A_{TM}$ then it shows that *w* is not accepted by *M*. As 0 is not acceptable by $M'$ therefore $L(M') = \{\langle M' \rangle\}$

Now show $\overline{A_{TM}}$ to $\overline{S}$. Consider another function *f* having input $\langle M, w \rangle$ and output $M''$ which works as follow on input *x*.

1. When $x == \langle M'' \rangle$

2. Execute $M$ on input *w*. If w is accept the **ACCEPT** else **REJECT**

3. **REJECT.**

It is also a many-one reduction from $\overline{A_{TM}}$ to $\overline{S}$. When $\langle M, w \rangle \notin A_{TM}$ then it shows that *w* is not accepted by *M*. According to $M''$ definition it can state that $M''$ is not acceptable by $M''$ therefore $M'' \notin S$. When $\langle M, w \rangle \in A_{TM}$ then it shows that *w* is accepted by *M* then according to $M''$ definition $M''$ is acceptable by $M''$.

Comment

## Problem

Let $R \subseteq \mathcal{N}^k$

A **a.** $R_0 = \{0\}$

**b.** $R_1 = \{1\}$

**c.** $R_= = \{(a,a) \mid a \in \mathcal{N}\}$

**d.** $R_< = \{(a,b) \mid a, b \in \mathcal{N} \text{ and } a < b\}$

## Step-by-step solution

**Step 1** of 5

The k-ary relation $R \subseteq N^k$ is ***definable*** in $Th(N,+)$, if a formula $\phi$ can be given with $k$ free variables $x_1,\ldots,x_k$, such that for all $a_1,\ldots,a_k \in N$, the formula $\phi(a_1,\ldots,a_k)$ is true only when $a_1,\ldots,a_k \in R$.

• The theory $Th(M)$ of a model $M$ is the collection of true sentences in the language of $M$.

• The theory for this problem is $Th(N,+)$, so the model will be $(N,+)$. Thus, only need to find formulas that are true for the given relations over the $(N,+)$ model.

Comment

**Step 2** of 5

a)

$R_0 = \{0\}$

• There is only one value $0$ to be considered. Adding this value to the variable $y \in N$ will produce no change in the value of $y$ as $y + 0 = y$.

• So, in the formula $\phi_0$ to make $R_0$ definable in $Th(N,+)$ for all $y$, it has; $\forall x \in R_0 \forall y[x + y \to 0 + y]$.

From this, the formula can be expressed as:

$\phi_0(x) = \forall y[x + y = y]$

Comment

**Step 3** of 5

b)

$R_1 = \{1\}$

• Similarly, in this case only one value is to be considered. So adding any value from $R_1 = \{1\}$ to the variable $y \in N$ will increment the value of $y$ by one as:

$\forall x \in R_1 \forall y[x + y \to 1 + y]$

Therefore, the formula $\phi_1$ to make $R_1$ definable in $Th(N,+)$ will be:

$\phi_1(x) = \forall y[x + y = y + 1]$

Comment

**Step 4** of 5

c)

$R_= = \{(a,a) \mid a \in N\}$

• The $R_=$ relation is a 2-ary relation. So, the formula to make it definable will be a binary function $\phi_=(x,y)$. This function has to be true if and only if $x = y$.

• For it be defined for $\forall z \in N$, the only possible value of $z$ is $0$. This is true when $\phi_0(z)$ is true, which can be written as $\forall z[\phi_0(z) \to x + z = y]$.

Hence, the function is given by:

$\phi_=(x,y) = \forall z[\phi_0(z) \to z + x = y]$

Comment

**Step 5** of 5

d)

$R_< = \{(a,b) \mid a,b \in N \text{ and } a < b\}$

• This relation is also 2-ary, so the formula will be a binary function $\phi_<(x,y)$. It needs to be true when $x < y$ and for all the values of the variable $\forall z \in N$.

• The only possible value of $z$ is $0$. Since all the other non-zero values will affect the magnitude of the quantity $x + z$, which will make it difficult to compare it against the value of $y$.

• As $\phi_0(z)$ is true when $z = 0$, it can be incorporated into $\phi_<$.

$\forall z[\phi_0(z) \to x + z = y]$.

This is the same as the definition of the function:

$\phi_<(x,y) = \forall z[\phi_0(z) \to z + x < y]$

Comment

## Problem

Answer each part TRUE or FALSE.

**a.** $2n = O(n)$.

**b.** $n^2 = O(n)$.

A **c.** $n^2 = O(n \log^2 n)$.

A **d.** $n \log n = O(n^2)$.

**e.** $3^n = 2^{O(n)}$.

**f.** $2^{2^n} = O(2^{2^n})$.

## Step-by-step solution

**Step 1** of 7

**TRUE (or) FALSE**

**Big – *O* Notation:**

Let *f* and *g* be functions $f, g : N \to R^+$ say that $f(n) = O(g(n))$ if positive integers *c* and $n_0$ exist such that for every integer $n \geq n_0$

$f(n) \leq c(g(n))$

When $f(n) = O(g(n))$ we say that $g(n)$ is an upper bound for $f(n)$.

Comment

**Step 2** of 7

(a)

**True.**

The statement $2n = O(n)$ is valid, because from the definition of Big-*O* notation it is clear that $f(n) = c(g(n))$.

Comment

**Step 3** of 7

(b)

**False.**

The statement $n^2 = O(n)$ is not valid, because $n^2 = n \cdot n$ which will grow faster than *n*.

That contradicts Big – *O* notation. Thus, $n^2 = O(n)$ is False.

Comment

**Step 4** of 7

(c)

**False.**

The statement $n^2 = O(n \log^2 n)$ is not valid, because factor *n* grows faster than the factor $\log^2 n$. That means $f(n) > g(n)$, which contracts the Big –*O* notation.

Hence $n^2 = O(n \log^2 n)$ is false.

Comment

**Step 5** of 7

(d)

**True.**

The statement $n\log n = O\left(n^2\right)$ is valid, because the factor $\log n$ grows slower than the factor *n*. That means $f(n) < g(n)$.From Big-*O* notation $n\log n - O\left(n^2\right)$ is true.

Comments (1)

**Step 6** of 7

(e)

**True.**

The statement $3^n = 2^{O(n)}$ is valid, because $3^n = 2^{n\log 3}$ and $n\log 3 = O(n)$.

From Big-*O* notation $3^n = 2^{O(n)}$ is true.

Comment

**Step 7** of 7

(f)

**True.**

The statement $2^{2^n} = O\left(2^{2^n}\right)$ is valid, because from Big- *O* notation $f(n) = O(f(n))$ for any function $f(n)$. Hence $2^{2^n} = O\left(2^{2^n}\right)$ is true.

Comment

## Problem

Answer each part TRUE or FALSE.

**a.** $n = o(2n)$.

**b.** $2n = o(n^2)$.

A **c.** $2^n = o(3^n)$.

A **d.** $1 = o(n)$.

**e.** $n = o(\log n)$.

**f.** $1 = o(1/n)$.

## Step-by-step solution

**Step 1** of 7

**TRUE (or) FALSE**

**Small – *o* Notation:**

Let *f* and *g* be functions $f, g : N \to R^+$ say that $f(n) = O(g(n))$ if for any real number

$c > 0$,a number $n_0$ exists, where *f(n) < c . g(n)* for all $n \geq n_0$.

Comment

**Step 2** of 7

(a)

**False.**

The statement $n = o(2n)$ is invalid, because the functions *n* and 2*n* grows equality

That is $f(n) = c \cdot g(n)$.But according to definition $f(n) < c \cdot g(n)$

Therefore $n = o(2n)$ is false

Comments (3)

**Step 3** of 7

(b)

**True.**

The statement $2n = o(n^2)$ is valid, because the functions $n^2 = n \cdot n$ which will grow faster than *n*. That is $f(n) < g(n)$.

Therefore from the definition of small – *o* notation, $2n = o(n^2)$ is true.

Comment

**Step 4** of 7

(c)

**True.**

The statement $2^n = o(3^n)$ is valid, because the function $2^n$ runs shower than

the function $3^n$ .

Then, $2^n < 3^n$.

That is $f(n) < c \cdot g(n)$

Hence from the definition of small – *o* notation, $2^n = o(3^n)$ is true.

Comment

**Step 5** of 7

(d)

**True.**

The statement $1 = o(n)$ is valid, because the function *n* grows faster than a number 1.

Therefore $f(n) < c \cdot g(n)$

By the definition of small – *o* notation, $1 = o(n)$ is true.

Comment

**Step 6** of 7

(e)

**False.**

The statement $n = o(\log n)$ is not valid, because the functions $\log n$ grows slower than the function *n* , which is a contradiction.

Hence $n = o(\log n)$ is false.

Comment

**Step 7** of 7

(f)

**False.**

The statement $1 = o(1/n)$ is not valid, because the function $1/n$ grows slower than 1.

Therefore $f(n) > g(n)$.

Which is a contradiction

Hence $1 = o(1/n)$ is false .

Comment

## Problem

Which of the following pairs of numbers are relatively prime? Show the calculations that led to your conclusions.

**a.** 1274 and 10505

**b.** 7289 and 8029

## Step-by-step solution

**Step 1** of 3

**Relative prime or not**

**Relative prime:** Two integers are said to be relative prime if only if common factor between them is 1. That is greatest common divisor $(GCD)$ is 1.

Comment

**Step 2** of 3

**(a)**

**The pair of numbers 1274 and 10505 are relatively prime.**

**Explanation:**

Take first $1274 = 2 \times 7 \times 7 \times 13 \times 1$.

Next take $10505 = 5 \times 11 \times 191 \times 1$.

Now observed, the common factor between 1274 and 10505 is 1.

That is, $GCD(1274, 10505) = 1$.

From the definition of relative prime, the GCD must have 1. Here GCD is 1.

So, the pair numbers 1274 and 10505 are relative prime.

Comment

**Step 3** of 3

**(b)**

**The pair of numbers 7289 and 8029 are not relatively prime.**

**Explanation:**

Take first $7289 = 37 \times 197 \times 1$.

Next take $8029 = 7 \times 31 \times 37 \times 1$.

Now observed, the common factors between 7289 and 8029 are 37 and 1.

Hence $GCD(7289, 8029) = 37$

From the definition of relative prime, the GCD must have 1. Here GCD is not equal to 1.

So, the pair numbers $7289$ and 8029 are not relatively prime.

Comment

## Problem

Fill out the table described in the polynomial time algorithm for context-free language recognition from Theorem 7.16 for string w = baba and CFG G:

$$S \to RT$$
$$R \to TR \mid \mathtt{a}$$
$$T \to TR \mid \mathtt{b}$$

THEOREM **7.16**

Every context-free language is a member of P.

## Step-by-step solution

**Step 1** of 3

**Table for a string w=baba :**

**Specified context free grammar** $(CFG)\ G$ **follows:**

$S \to RT$
$R \to TR \mid a$
$T \to TR \mid b$

Comment

**Step 2** of 3

Take string $w = baba$

Now we will construct the table for string baba:

Initially the table will be as shown below:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| string | b | a | b | a |

First all the substring string which are having rules of the $A \to b$ will be occupied in the location where i =j. As per the theorem.

So the table will be filled as follows:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | $T$ | | | |
| 2 | | $R$ | | |
| 3 | | | $T$ | |
| 4 | | | | $R$ |
| string | b | a | b | a |

Now fill the remaining entries of the table such that $i \le j$ and substring formed by one or more substring such that rule $A \to BC$.

Consider the string *ba* for its suitable replacement rule.

There is no rule for this string. So the variables of the corresponding individual strings are added to the table entry.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | $T$ | $R,T$ | | |
| 2 | | $R$ | | |
| 3 | | | $T$ | |
| 4 | | | | $R$ |
| string | b | a | b | a |

Consider the string *ab* for its suitable replacement rule.

Similarly fill the next entry in the same row with rule $S \to RT$. Since it is producing the sting *ab.*

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | $T$ | $R,T$ | | |
| 2 | | $R$ | $S$ | |
| 3 | | | $T$ | |
| 4 | | | | $R$ |
| string | b | a | b | a |

Similarly the remaining entries of the table are filled as follows using the above 2 conditions of the algorithm as the proof for the Theorem 7.16:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | $T$ | $R,T$ | | |
| 2 | | $R$ | $S$ | |
| 3 | | | $T$ | $R,T$ |
| 4 | | | | $R$ |
| string | b | a | b | a |

Comment

**Step 3** of 3

Now the two diagonals are filled and using the same rules the third diagonal is also filled.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | $T$ | $R,T$ | $S$ | |
| 2 | | $R$ | $S$ | $R,T$ |
| 3 | | | $T$ | $R,T$ |
| 4 | | | | $R$ |
| string | b | a | b | a |

Similarly fill left over entry in the table using the conditions mentioned in the algorithm.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | $T$ | $R,T$ | $S$ | $S$ |
| 2 | | $R$ | $S$ | $R,T$ |
| 3 | | | $T$ | $R,T$ |
| 4 | | | | $R$ |
| string | b | a | b | a |

Comments (5)

## Problem

Is the following formula satisfiable?

$$(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$$

## Step-by-step solution

**Step 1** of 3

**The formula** $(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$ **is not satisfiable**.

**Explanation:**

• A Boolean formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

• We know that

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \overline{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1. & 1 \vee 1 = 1 & \end{array}$$

Comment

**Step 2** of 3

Specified Boolean formula is

$$(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$$

Here consider *x* and *y* are variables.

**Case 1:**

Assign $x = 0$ and $y = 1$

Then $(0 \vee 1) \wedge (0 \vee \overline{1}) \wedge (\overline{0} \vee 1) \wedge (\overline{0} \vee 1) \wedge (\overline{0} \vee \overline{1})$

$$\begin{aligned} &= 1 \wedge (0 \vee 0) \wedge (1 \vee 1) \wedge (1 \vee 0) \\ &= 1 \wedge 0 \wedge 1 \wedge 1 \\ &= 0 \wedge 1 \\ &= 0 \end{aligned}$$

Comment

**Step 3** of 3

**Case 2:**

Assign $x = 1$ and $y = 0$

Then $(1 \vee 0) \wedge (1 \vee \overline{0}) \wedge (\overline{1} \vee 0) \wedge (\overline{1} \vee 0) \wedge (\overline{1} \vee \overline{0})$

$$\begin{aligned} &= 1 \wedge (1 \vee 1) \wedge (0 \vee 0) \wedge (0 \vee 1) \\ &= 1 \wedge 1 \wedge 0 \wedge 1 \\ &= 1 \wedge 0 \\ &= 0 \end{aligned}$$

From case 1 and case 2 of the Boolean values for *x* and *y*, the formula always evaluated to 0, but we know that if some assignment of 0s and 1s to the variables makes the formula evaluate to 1 then a Boolean formula is satisfiable.

So, the formula $(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$ is not satisfiable.

Comment

## Problem

Show that P is closed under union, concatenation, and complement.

## Step-by-step solution

**Step 1** of 4

**The class P is closed under union, concatenation, and complement**

**Class P:** P is the class of languages that are decidable in polynomial time on a deterministic single – tape Turing machine.

That is, $P = \bigcup_k \mathrm{TIME}\left(n^k\right)$

Now we have to show that P is closed under union, concatenation and complement.

Comment

**Step 2** of 4

**Union:**

Assume two languages $P_1 \in P$ and $P_2 \in P$

The Turing machine *M* that accepts $P_1 \cup P_2$ works as follows:

$M =$ "On input *w*:

1. Check if $w \in P_1$

2. if not then check if $w \in P_2$

3. Accept *w* if and only if $P_1$ or $P_2$ accepts.

4. If both reject then rejects the input *w*".

Since each membership check requires polynomial time the overall time is polynomial.

Comment

**Step 3** of 4

**Concatenation:**

Assume two languages $P_1 \in P$ and $P_2 \in P$.

The Turing machine *M* that accepts $P_1.P_2$ works as follows

$M =$ "On input *w* of length *n:*

1. *w* can be split into two strings in *n* different ways

2. For each split,

(a) Check if the first substring belongs to $P_1$

(b) check if second substring belongs to $P_2$

3. If any split succeeds, then accept".

Clearly the overall time is polynomial.

Comments (1)

**Step 4** of 4

**Complement :**

Assume a language $P_1 \in P$

The Turing machine *M* that accepts $\overline{P_1}$ works as follows

$M =$ "On input *w*:

1. Check if $w \in P_1$

2. If $w \in P_1$ then reject.

3. If $w \notin P_1$, then accept"

Clearly the overall time is polynomial.

Therefore, the class P is closed under union, concatenation, and complement.

Comments (1)

## Problem

Show that NP is closed under union and concatenation.

## Step-by-step solution

**Step 1** of 3

**The class NP is closed under union and concatenation**

***NP* – class:**

*NP* is a class of languages that are decidable in nondeterministic polynomial time on a non – deterministic Turing machine.

Comment

**Step 2** of 3

**Union:**

• Let *A* and *B* be languages are decided by $NP-$machines $T_A$ and $T_B$.

• Now we want to show that, there is a non – deterministic poly time decider $T_{A\cup B}$ that decides union of *A* and *B.*

• The construction of $T_{A\cup B}$ is as follows:

$T_{A\cup B} =$" On input *S*:

1. Run $T_A$ on *S*. If $T_A$ accepts *S,* then accept.

2. Else run $T_B$ on *S*. If $T_B$ accepts *S,* then accept.

3. Else reject"

As the new *TM* $T_{A\cup B}$ calls $T_A$ and $T_B$ each once, it runs on $O(T_A+T_B)$,

as both are *NP* is $T_{A\cup B}$.

Comment

**Step 3** of 3

**Concatenation:**

• Let *A* and *B* be languages are decided by $NP-$machines $T_A$ and $T_B$.

• Now we want to show that, there is a non – deterministic poly time decider $T_{A\cup B}$ that decides concatenation of *A* and *B.*

• The construction of $T_{A\circ B}$ is as follows:

$T_{A\circ B}$ = "On input S:

1. Split S into $S_1$, $S_2$ such that $S=S_1S_2$.

2. Run the NP machine $T_A$ on $S_1$. If $T_A$ is rejected, then reject.

3. Else run $T_B$ on $S_2$. If $T_B$ is rejected, then reject.

4. Else accept.

The time taken by step 1 is $\mathrm{O}(n)$ in a two tape Turing Machine. Thus, T is a poly-time non-deterministic decider for $A\circ B$.

Comments (1)

## Problem

Let *CONNECTED* = { $\langle G \rangle$ | G is a connected undirected graph}. Analyze the algorithm given on page 185 to show that this language is in P.

## Step-by-step solution

**Step 1** of 1

Consider the algorithm given on the page 185 in the textbook. A problem is said to be in *P* when method for solving the problem runs in time $n^k$ where k is a constant.

• In the first line, first node is selected and marked. This will require a constant time.

• In second line and third line, the nodes are marked until all the nodes are marked. If there are *n* nodes, then scanning of the nodes is there from the list 2, 3, 4, . . . . . up to *n-1*. For every node, the order of neighbors will be order of *n*. This will result into the order of $n^3$ inspections.

• In line 4, all the nodes are scanned for determining whether all the nodes are marked or not. This will take time equal to order of *n*.

As the order is equal to $n^3$, so this algorithm belongs to *P*.

Comments (1)

## Problem

A ***triangle*** in an undirected graph is a 3-clique. Show that *TRIANGLE* ? P, where *TRIANGLE* = { $\langle G \rangle$ I G contains a triangle}.

## Step-by-step solution

**Step 1** of 1

Class *P* : *P* is a class of languages that are decidable in polynomial time on a deterministic single – tape Turing – machine.

Specified language,

• A triangle is an undirected graph is a 3 – clique.

• TRIANGLE $= \{\langle G \rangle \mid G \text{ contains a triangle}\}$.

• Now we have to show that TRIANGLE $\in p$

• Let *A* be the Turing machine that decides TRIANGLE is polynomial time

• *A* can be described as follows:

*A*="on input $G\langle V, E \rangle$:

*V* denotes set of vertices of the graph *G*.

*E* denotes set of edges of the graph *G*.

1. For $u, v, w \in V$ and $u < v < w$, we enumerate all triples $\langle u, v, w \rangle$.

2. Check whether all three edges $(u, v), (v, w)$ and $(w, u)$ exist in *E* or not. If exist then accept.

3. Otherwise reject."

• Enumeration of all triple require $o\left(|v|^3\right)$ time

• Checking whether all three edges belong to *E* take $O(|E|)$ time.

• Overall time is $O\left(|V|^3 |E|\right)$ which is polynomial in the length of the inplet

• Therefore TRIANGLE $\in P$

Comment

## Problem

Show that $ALL_{DFA}$ is in P.

## Step-by-step solution

**Step 1** of 2

Class $P$ : $P$ is a class of languages that are decidable in polynomial time on a deterministic single – tape Turing – machine.

$\text{All}_{DFA} = \{\langle A \rangle \; A \text{ is a } DFA \text{ that recognies } \Sigma^*\}$

Comment

**Step 2** of 2

$E_{DFA} = \{\langle A \rangle \; A \text{ is a } DFA \text{ and } L(A) = \phi\}$ is determined by a Turing – machine $(TM)$

Let $E$ be the Turing machine that determines $E_{DFA}$

Let $R$ be the Turing machine that determines $ALL_{DFA}$

The algorithm of $R$ is as follows:

$R = \text{"On input } \langle A \rangle, \text{ where } A \text{ is a } DFA$ :

1. Construct a $DFA$ $B$ that recognizes $L(\overline{A})$, by swapping accept and non – accepting states

2. Run the $TM$ $E$ on input $\langle B \rangle$, where $E$ determines $E_{DFA}$.

3. If $E$ accepts, then accept

4. If $E$ rejects, then reject."

Clearly the $TM$, $R$ determines $ALL_{DFA}$ in polynomial time.

Therefore, $ALL_{DFA}$ is in $P$.

Comments (4)

## Problem

In both parts, provide an analysis of the time complexity of your algorithm.

**a.** Show that $EQ_{DFA}$ ? P.

**b.** Say that a language A is ***star-closed*** if A = $A^*$. Give a polynomial time algorithm to test whether a DFA recognizes a star-closed language. (Note that $EQ_{NFA}$ is not known to be in P.)

## Step-by-step solution

**Step 1** of 6

The given details are as follows:

$EQ_{DFA} \in P$

Where *P* is a class of language and *EQ* represents a deterministic finite automata (*DFA*).

Comment

**Step 2** of 6

Consider *E* and $Q$ be two $DFA$, then a third $DFA$ $C$ can be constructed, which is defined as follows:

$L(C) = \left(L(E) \cap \overline{L(Q)}\right) \cup \left(L(Q) \cap \overline{L(E)}\right)$.

Here, first the intersection of the language recognized by the *DFA E* and the complement of the language recognized by the *DFA Q* takes place and then its union takes place with the intersection of the language recognized by the *DFA Q* and the complement of the language recognized by the *DFA E*.

• In the process of *DFA* $C$ construction, interchanging in accepting state can be used for complement operation.

• It is already known that, polynomial time can be taken in intersection and union operations. So, it can be said that "the polynomial time required finishing all the construction procedure".

Comment

**Step 3** of 6

**Algorithm:**

**L="On input ":**

1. **w will be run by L**

2. **If** $L(C) = \varnothing$ **(NULL) or** $L(E) = L(Q)$ **is accepted.**

**Else it is rejected**.

Comment

**Step 4** of 6

Now testing is performed on $L(C)$ and check that where it becomes NULL. In other words, $L(C) = \varnothing$ or $L(E) = L(Q)$.

• Now suppose, $k$ = the number of states of $C$ (polynomial size of $E$ and $Q$ ), then it is already known that "the number of step 2 $\leq n$ and every step 2 takes a cost less than $n^2$. Hence, it can be said that," $L(C) = \varnothing$ **is polynomial**".

• Therefore, **a Turing machine is obtained which decide a polynomial time is used to run** $EQ_{DFA}$**.** In other words, it can be said that**,** " $EQ_{DFA} \in P$ **"**.

• Hence, the time complexity of the algorithm is $O(n^2)$ because both *DFA E* and *Q* take $O(n)$ time so, the total time is $O(n) \times O(n) = O(n^2)$.

Comment

**Step 5** of 6

**b)**

A language $A$ is known as star-closed if $A = A^*$. An algorithm can be provided, which is used to **test whether a *DFA* recognizes a star-closed language**. It can be achieved by the following algorithm:

• For a given language $A$, suppose T be the Turing machine which decides it in polynomial in time.

• Now a Turing machine T' is constructed which is used to decide the star closed of the given language $A$ in polynomial time.

**T'="On input ~~".~~**

~~**1. s will be run by T**~~

~~**2. If T is accepted, reject. Else it is rejected, accepted."**~~

Comments (1)

**Step 6** of 6

**Here, T' is used to decide the star of the language $A$. Since T runs on polynomial time, then T' will also be run on polynomial time**.

Therefore, the above describes that "**DFA recognized a star-closed language"**.

The input string ***s*** for a language ***A*** can be ( $\varepsilon$ ) or an input for any number of times possible. Let, the input string ***k***-times possible, thus for each input time complexities are different and consider upper bound time $O(1) \times O(2) \times ..... \times O(n^k) = O(n^k)$.

So, the total time complexity for *k* times is $O(n^k)$.

Comment

## Problem

Call graphs *G* and *H* ***isomorphic*** if the nodes of *G* may be reordered so that it is identical to *H*. Let *ISO* = { $\langle G, H \rangle$ | G and H are isomorphic graphs}. Show that *ISO* ? NP.

## Step-by-step solution

**Step 1** of 4

**Class *NP* :**

*NP* is a class of languages that are nondeterministic polynomial time on a non – deterministic single – tape Turing Machine.

From the definition 7.19 *NP* is the class of languages that have polynomial time verifies

Consider the given expression:

ISO $= \{\langle G, H \rangle \mid G \text{ and } H \text{ are isomorphic graphs}\}$

• If the nodes of *G* may be reordered so that it is identical to *H* then Graphs *G* and *H* are said to be isomorphic.

• Now it must be proved that $ISO \in NP$

• Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be the two graphs

• Let $V_G = \{u_1, u_2, \ldots u_m\}, V_H = \{v_1, v_2, \ldots v_n\}$ be the sets of vertices of *G* and *H.*

Comment

**Step 2** of 4

**Isomorphism:**

An isomorphism is defined by a mapping $f : V_G \to V_H$ with the property that it is a one – to –one correspondence. That means it is both one – to – one and onto.

• This one – to – one correspondence is possible only if $m = n$ and for all $u, v \in V_G$ we have $(v, v) \in E_G$ if and only if $(f(u), f(v)) \in E_H$.

• Thus, the correspondence takes edges into edges and non – edges into non – edges.

• A mapping *f* can be represented. By a sequence $S = (S_1, S_2, \ldots S_m)$ of indices with the property that $f(u_i) = v_{s_i}$, that is $i^{th}$ point of *G* is mapped into the $S_i^{th}$ point of *H.*

• This sequence *S* can be taken as certificate.

Comment

**Step 3** of 4

Now *N* is the non – deterministic Turing machine $(NTM)$ that decides *ISO* in polynomial time.

$N =$ "On input $(\langle G, H \rangle, S)$:

Where *G* and *H* are graph as defined above *S* is the certificate.

1. Check whether *G* and *H* have same number of points.

2. If *G* and *H* have same number of points then checks that for each pair $i, j$

$$\Rightarrow \left(v_{S_i}, v_{S_j}\right) \in E_U \quad .....(1)$$
$$\Rightarrow \left(u_i, u_j\right) \in E_U \quad .....(2)$$
$$From\ (1)\ and\ (2)$$

i. $E_U$ can be derived from the above mapping procedure, $f\left(u_i\right) = v_{S_i} = E_U$

ii. have $S_i \neq S_j$ and that $\left(u_i, u_j\right) \in E_U$ if and only if $\left(v_{S_i}, v_{S_j}\right) \in E_U$

iii. If the above condition satisfies, then “accept”.

3. Otherwise “reject”.

Comment

**Step 4** of 4

All these checking can be done in time $O\left(m^2\right)$, so in time polynomial in the description of $\left(G, H\right)$. **Therefore** $ISO \in NP$**.**

Comment

## Problem

Let

$$MODEXP = \{\langle a, b, c, p\rangle \mid a, b, c, \text{ and } p \text{ are positive binary integers such that } a^b \equiv c \pmod{p}\}.$$

Show that *MODEXP* ? P. (Note that the most obvious algorithm doesn't run in polynomial time. Hint: Try it first where b is a power of 2.)

## Step-by-step solution

**Step 1** of 2

Consider,

$$\text{MODEX}P = \left\{ \begin{array}{l} \langle a,b,c,p\rangle \mid a,b,c \text{ and } p \text{ are binary integers} \\ \text{such that } a^b \equiv c \pmod{p} \end{array} \right\}$$

Comment

**Step 2** of 2

A polynomial time algorithm *M* for MODEXP is as follows:

$M =$ "On input $< a,\ b,\ c, p >$, where $a,\ b,\ c$ and $p$ are binary integers.

- Calculate $x = a \bmod p$, initialize $y$ to 1 and $i$ to 0.
- For $b = b_n b_{n-1} \ldots b_1 b_0$, do the following $n+1$ times:
- if $b_i = 1$, then $y = y \cdot x \bmod p; x = x^2 \bmod p; i = i+1$
- if $y \equiv c (\bmod p)$, accept. Otherwise, reject."

The algorithm runs in polynomial time. In the above algorithm, steps 1 and 4 will be executed once. The step 3 needs $O(n)$ time. Thus, *M* is a polynomial time algorithm for *MODEXP*.

**Therefore, *MODEXP* $\in$ *P*.**

Comment

## Problem

A ***permutation*** on the set {1, . . . , k} is a one-to-one, onto function on this set. When p is a permutation, $p^t$ means the composition of p with itself t times. Let

$$PERM\text{-}POWER = \{\langle p, q, t\rangle \mid p = q^t \text{ where } p \text{ and } q \text{ are permutations on } \{1, \ldots, k\} \text{ and } t \text{ is a binary integer}\}.$$

Show that *PERM-POWER* ? P. (Note that the most obvious algorithm doesn't run within polynomial time. Hint: First try it where t is a power of 2.)

## Step-by-step solution

**Step 1** of 2

A permutation on the set $\{1, 2, \ldots, k\}$ is a one-to-one, onto function on this set. If *p* is a permutation then $p^t$ says that the composition of *p* with itself $t$ times.

The $\text{PERM-POWER}$ is defined as follows:

$$\text{PERM-POWER} = \{< p, q, t > \mid p = q^t \text{ where } p \text{ and } q \text{ are permutations on } \{1, \ldots, k\} \text{ and } t \text{ is a binary integer}\}$$

Comment

**Step 2** of 2

The binary integer *t* can be represented as $t = x_0 2^0 + x_1 2^1 + \ldots + x_n 2^n$ where $x_i$ acquires a value either 0 or 1.

Now, $q^t$ can be written as,

$$\begin{aligned} q^t &= q^{x_0 2^0 + x_1 2^1 + \ldots + x_n 2^n} \\ &= q^{x_0 2^0} \times q^{x_1 2^1} \times \ldots \times q^{x_n 2^n} \end{aligned}$$

From this, compute $q^{2^j}$ where $j = 1, 2, \ldots, \lfloor \log t \rfloor$. By substituting *j* value, $q^{2^j}$ can be $q^1, q^2, q^4, q^8, \cdots$. It is easy to compute the permutation by applying *q* on *q* itself. It takes $O(k \log t)$ steps to compute $q^{2^j}$ where each product requires $O(k)$ steps. Finally, the value of $q^{2^j}$ is compared with *p* which takes additional *k* steps. Thus, it can be said that $\text{PERM-POWER} \in \text{P}$.

Comment

## Problem

Show that P is closed under the star operation. (Hint: Use dynamic programming. On input $y = y_1 \cdots y_n$ for $y_i$ ? $\Sigma$, build a table indicating for each $i \leq j$ whether the substring $y_i \cdots y_j$ ? $A^*$ for any A ? P.)

## Step-by-step solution

**Step 1** of 2

Class *P* : *P* is a class of languages that are decidable in polynomial time on a deterministic single tape Turing machine. That is $P = \bigcup_k \mathrm{TIME}(n^k)$.

Comments (4)

**Step 2** of 2

Now, prove that *P* is closed under star operation. Consider a language $A \in P$. The following procedure decides *A**:

$M =$ "*On input assume* $y = y_1 y_2 \ldots y_n \in \Sigma$.

1) if $y = \varepsilon$, output ACCEPT and halt.
2) Initialize the table $T[i, j] = 0$ for $i \leq j$.
3) For $i = 1$ to $n$
   a) Run $M$ on $y_i$, *if* $y_i \in L$ then set $T[i, i] = 1$.
4) For $k = 2$ to $n$
   For $i = 1$ to $n - k + 1$
   a) Assume $j = i + k - 1$
   b) Run $M$ on $y_i \ldots y_j$, *if* $y_i \ldots y_j \in L$ then set $T[i, j] = 1$.
   c) For $l = i$ to $j - 1$
      Set $T[i, j] = 1$, *if* $T[i, l] = 1$ and $T[l, j] = 1$
5) Output ACCEPT if $T[1, n] = 1$; otherwise output REJECT.

**The above algorithm is a polynomial time algorithm. Therefore, *P* is closed under star operation.**

Comments (2)

## Problem

Show that NP is closed under the star operation.

## Step-by-step solution

**Step 1** of 1

Class- *P* : *P* is a class of languages that are decidable in polynomial time on a deterministic single – tape Turing machine.

Now we have to prove that *NP* is closed under star operation.

• Let *L* be the language that decided by *NP* – machine

• Now we will construct a non deterministic Turing machine *NTIM N* to decide $L^*$ in nondeterministic polynomial time

• Construction of *N* is as follows:

$N =$ "On input *w*:

1. If $w = \in$ then accept.

2. Non-deterministically slipt *w* into *k* pieces $w = w_1 w_2 \ldots w_k$

3. For each $w_i$, non-deterministically guess the certificates that show $w_i \in L$

4. Verify all certificates

(a) If verification is done then accept.

(b) else if verification is fails, then reject."

Thus we constructed a *NTM N* that decides $L^*$ in nondeterministic polynomial time.

Hence *NP* is closed under star operation.

Comment

## Problem

Let *UNARY-SSUM* be the subset sum problem in which all numbers are represented in unary. Why does the NP completeness proof for *SUBSET-SUM* fail to show *UNARY-SSUM* is NP-complete? Show that *UNARY-SSUM* ? P.

## Step-by-step solution

**Step 1** of 2

Let *UNARY-SSUM* be the subset sum problem in which all numbers are represented in unary.

The complexity of a function can be measured from the input size and number of steps required for that input.

• If the input is in unary, then the size of the input is equal to the value. When the input value is *x*, then number of squares needed to represent the input is *x*.

• If the input is in binary, then log *x* squares needed to represent the input value *x*. Thus, if the input size in binary is *N*, then the input size in unary is $2^N$.

• It can be concluded that the size of the input is increasing exponentially.

• The reduction proof requires more than polynomial time to reduce 3*SAT* to *UNARY-SSUM*.

NP-completeness proof for *SUBSET-SUM* fails to show *UNARY-SSUM* is NP-complete because, the reduction is not polynomial time. The input size in unary is $2^N$ which takes exponential time.

Therefore, the NP-completeness proof for *SUBSET-SUM* fails to show *UNARY-SSUM* is NP-complete.

Comment

**Step 2** of 2

To prove the language is in P, need to show that the polynomial time algorithm decides it. The standard algorithm for *SUBSET-SUM* runs in the time bounded by a polynomial in the input size *N* and the target integer *S*. The polynomial time algorithm for *UNARY-SSUM* is as follows:

• Compute the sum of numbers in the input.

• Check if the sum of the numbers is at least the target integer *S*.

• If the sum of the numbers is at least the target integer *S*, then run the standard dynamic programming algorithm of *SUBSET-SUM*.

• Otherwise, reject.

There exists a polynomial time algorithm to decide *UNARY-SSUM*. Therefore, $UNARY - SSUM \in \mathrm{P}$.

Comment

## Problem

Show that if P = NP, then every language A ? P, except A = ø and A = $\Sigma^*$, is NP-complete.

## Step-by-step solution

**Step 1** of 2

A language B is said to be NP-complete if the following conditions are satisfied:

1. $B \in NP$

2. Every language L can be polynomial-time reduced to B.

Comment

**Step 2** of 2

Let P = NP and let $A \in P$ such that $A \neq \phi$ and $A \neq \Sigma^*$ so, it is required to prove that for every $A \in NP$ and $L \in NP$, $L \leq_P A$.

Let there exist an arbitrary language L from NP=P. Hence, the language L has polynomial decider $X_L$ so, the polynomial reduction f from L to A will be as follows:

When the input string is x:

1. Run $X_L$ on the string x.

2. If the decider $X_L$ accepts the string then output $x_{in}$

3. If the decider $X_L$ rejects the string then output $x_{output}$

Thus, there exists a poly-time reduction from L to A, so, A is NP-complete.

Comment

## Problem

Show that *PRIMES* = {ml m is a prime number in binary} ? NP. (Hint: For p > 1, the multiplicative group $Z^*_p$ = {xl x is relatively prime to p and 1 ≤ x < p} is both cyclic and of order p – 1 iff p is prime. You may use this fact without justifying it. The stronger statement *PRIMES* ? P is now known to be true, but it is more difficult to prove.)

## Step-by-step solution

**Step 1** of 2

Here, $\mathbf{PRIMES\ (P) = \{m \mid m \text{ is a prime number in binary}\} \in NP}$ can be proved by the following approach.

Now, consider two situations:

1. Consider a situation where $P>1$: (Because all prime numbers are greater than 1).

2. The multiplicative group $Z^*_P = \{X|X \text{ is relatively prime to P and } 1 \le x \le P\}$.

Comment

**Step 2** of 2

Here, a situation is considered where *x* is a relative prime number the value of $x$ lies between 0 and 1.

• Both conditions are cyclic as the both situations can be combined and can lie between 1 and P.

• **Order of these conditions is** $P\text{-}1$ **if** $P$ **is prime as the range lies between 1 and P then the order between P-1. This fact is alone sufficient to prove the statement** $\text{PRIMES (P)} = \{\text{m| m is a prime number in binary}\} \in \text{NP}$ and second considered situation are quite enough itself to justify the statement.

• It can be proved by **the fact of belonging of prime numbers to** $\text{co-NP}$ **and consider prime numbers belong to** $\text{co-RP}$**.**

• **Thus, required statement will be true as well, because belongingness of NP can be proved only if belongs to** $\text{co-NP}$ **and** $\text{co-RP}$ **as well.**

This it is quite obvious that, $\mathbf{PRIMES\ (P) = \{m \mid m \text{ is a prime number in binary}\} \in NP}$.

Comment

## Problem

We generally believe that *PATH* is not NP-complete. Explain the reason behind this belief. Show that proving *PATH* is not NP-complete would prove P ≠ NP

## Step-by-step solution

**Step 1** of 2

***NP* – complete:**

A language *B* is *NP* – complete if it satisfies two conditions

1. *B* is in *NP* and

2. Every *A* is *NP* is polynomial time reducible to *B*.

Comment

**Step 2** of 2

PATH $= \{\langle G,s,t\rangle \mid G \text{ is a directed graph that has a directed path from s to t}\}$

1. **PATH is not *NP* – complete:**

◊ Let us assume that PATH would be *NP* – complete.

◊ From the definition of *NP* – completeness,

For all $L \in NP, L$ is polynomial time reducible to PATH.

◊ But this again implies that for all *L* in $NP, L$ is in *P*. Thus $P = NP$ which we believe that it is not true.

**Hence, PATH is not *NP* – complete.**

2. Proving that PATH is not *NP* – complete implies that $NP \neq P$:

◊ Showing this by contraposition.

◊ Assume that $P = NP$ and then show that PATH is *NP* – complete.

◊ So assume $P = NP$.

◊ "If $P = NP$ then every language $A \in P$ is *NP* – complete".So, PATH is *NP* – complete.

**Thus, if PATH is not *NP* –complete, then $NP \neq P$ .**

Comment

## Problem

Let G represent an undirected graph. Also let

$$SPATH = \{\langle G, a, b, k\rangle \mid G \text{ contains a simple path of length at most } k \text{ from } a \text{ to } b\},$$

and

$$LPATH = \{\langle G, a, b, k\rangle \mid G \text{ contains a simple path of length at least } k \text{ from } a \text{ to } b\}.$$

**a.** Show that *SPATH* ? P.

**b.** Show that *LPATH* is NP-complete.

## Step-by-step solution

**Step 1** of 3

a)

**Class- *P* :** *P* is class of languages that are decidable in polynomial time on a deterministic single – tape Turing machine. We have to construct an deterministic Turing machine $(DTM)$ to decide *SPATH* in polynomial time.

Let *M* be the *DTM* to decide *SPATH* in polynomial time.

The algorithm of *M* is as follows:

$M =$"on input $\langle G,a,b,k\rangle$ where *m*-node graph *G* has nodes *a* and b:

1. Place a mark "o" on node *a*.

2. for each *i* from 0 to *m*:

3. If an edge $(s,t)$ is found connecting *s* marked as "*i*" to an unmarked node *t*, mark node *t* with $"i+1"$.

4. If *b* is marked with a value at most *k*, accept. Otherwise reject.

This algorithm is similar to PATH algorithm. Here we additionally need to keep the track of length of the shortest paths discovered. That will be done in polynomial time $O(|V|+|E|)$.

Hence, we constructed a *DTM M* to decide *SPATH* in polynomial time.

Therefore, $SPATH + 1 \in P$.

Comments (1)

**Step 2** of 3

(b)

***NP* - complete:** A language *B* is *NP* – complete if it satisfies two conditions.

1. *B* is in *NP* and

2. Every *A* in *NP* is polynomial time reducible to *B*.

To show *LPATH* is *NP* – complete, we need show $LPATH \in NP$ and $UHAMPATH \leq_p LPATH$

**1.** $LPATH \in NP$ **:**

We know that "*NP* is the class of languages that have polynomial time verifies.

We construct a verifies *V* for $LPATH$ as follows:

$V =$" input $\langle G,a,b,k,c \rangle$, where *c* is a path:

1. Check *c* is a non – repeated sequence of nodes in *G*.

2. Check the first term of *c* is a and last is *b*.

3. Check the length of *c* is larger than or equal to *k*.

4. If *c* satisfies the conditions 1 to 3, accept.

5. Otherwise, reject

This verifier *V* can finish in $O(ICI)$ where $|c|$ is the length of *c*.

**So,** $LPATH \in NP$**.**

Comments (1)

**Step 3** of 3

2. $UHAMPATH \leq_p LPATH$ :

Consider an instant $\langle G,a,b \rangle$ of $UHAMPATH$ problem where $G = \langle V,E \rangle$ is a graph with assigned starting node *a* and ending node *b*.

• The mapping copy $\langle G,a,b \rangle$ and set $k = |V| - 1$, then $\langle G,a,b,k \rangle$ is an instance of $LPATH$ .

• It can be finished in polynomial time $(O(|V| + |E|))$

• We need to prove $\langle G,a,b \rangle \in UHAMPATH \Leftrightarrow \langle G,a,b,k \rangle \in LPATH$

If $\langle G,a,b \rangle \in UHAMPATH$ , then *G* has a Hamiltonian path from *a* to *b*.

• It must be a simple path that goes through every node exactly once,

• Which implies that the length is $|v| - 1 = k$

• So $\langle G,a,b,k \rangle \in LPATH$

If $\langle G,a,b,k \rangle \in LPATH$ , there exists a simple path from *a* and *b* with length $k = |v| - 1$.

• Because the graph *G* only has $k+1$ nodes.

• So this simple path must pass through all of needs in graph *G* exactly once.

• So this simple path must be a Hamiltonian path.

• It implies that $\langle G,a,b \rangle \in UHAMPATH$

**Therefore, the** $LPATH$ **is *NP* – complete.**

Comment

## Problem

Let *DOUBLE-SAT* = { $\langle \phi \rangle \mid \phi$ has at least two satisfying assignments}. Show that *DOUBLE-SAT* is NP-complete.

## Step-by-step solution

**Step 1** of 3

***NP* –complete:** A language *B* is *NP* – complete if it satisfies two conditions.

1.*B* is in *NP* and

2. Every *A* in *NP* is polynomial time reducible to *B*.

Comment

**Step 2** of 3

**1.** $DOUBLE-SAT \in NP$

NTM *N* can decide double – *SAT* as follows:

$N =$" on input a Boolean formula $\phi$:

1. Non-deterministically guess two Boolean assignments $t_1$ and $t_2$ which are different from each other.

2. If both $t_1$ and $t_2$ satisfies $\phi$ then accept

3. Otherwise, reject.

Thus, we construct a $\text{NTM } N$ to decide Double – *SAT*.

Hence, $DOUBLE-SAT \in NP$.

Comments (1)

**Step 3** of 3

2. $SAT \leq_p DOUBLE-SAT$

• The function *f* which maps an instance $\phi$ of *SAT* to an instance $\phi'$ of $DOUBLE-SAT$ work as follows:

$\phi' = \phi \wedge (x_1 \vee x_2)$

• Where $x_1$ and $x_2$ are new variables. They do not occur in $\phi$.

• This reduction is certainly polynomial time.

• If $\phi$ is unsatifiable, certainly $\phi'$ is also unsatisfiable. Because we have only conducted an additional term.

• But if $\phi$ has some satisfying assignments $t$, then $\phi'$ has at least three satisfying assignments, corresponding to the 3 different ways of extending *t* to the new variables $x_1$ and $x_2$.

**Thus, the $DUBLE-SAT$ is *NP* - complete.**

Comment

## Problem

Let *HALF-CLIQUE* = { $\langle G \rangle$ | G is an undirected graph having a complete subgraph with at least m/2 nodes, where m is the number of nodes in G}. Show that *HALF-CLIQUE* is NP-complete.

## Step-by-step solution

**Step 1** of 2

Clique is an undirected graph where every two nodes connected by an edge.

***NP* - complete:**

A language *B* is *NP* – complete if by an edge it satisfies two conditions.

1. *B* is in *NP*

2. Every *A* in *NP* is polynomial time reducible to *B*.

Comment

**Step 2** of 2

1. $HALF-CLIQUE \in NP$:

Let *N* be the nondeterministic polynomial time (*NTM)* that decides $HALF-CLIQUE$ in polynomial time.

*N* can be described as follows:

$N =$"on input graph $\langle G \rangle$:

1. Non-deterministically choose at least $n/2$ nodes

2. Verify whether $n/2$ nodes form a clique

3. If they form a clique then accept.

4. Otherwise, reject".

**Therefore,** $HALF-CLIQUE \in NP$

2. $CLIQUE \leq_p HALF-CLIQUE$:

A reduction from $CLIQUE$ to $HALF-CLIQUE$ as follows:

On input $\langle G,k \rangle$, where *G* is a graph on *n* verifies and *k* is an integer:

1. If $k = n/2$ then output $\langle G \rangle$.

2. If $k < n/2$, then construct a new graph $G'$ by adding a complete graph with $n-2k$ vertices and connecting them to all vertices in *G*, and output $\langle G' \rangle$.

3. If $k > n/2$, then construct a new graph $G''$ by adding $2k-n$ isolated vertices to *G*, and output $\langle G'' \rangle$.

$When\, k = n/2$:It is clear that $\langle G, n/2 \rangle \in CLIQUE$ if and only if $\langle G \rangle \in HALF-CLIQUE$.

When $k < n/2$:If *G* has a *k*-clique, then $G'$ has a clique of size

$k + (n-2k) = (2n-2k)/2$.

Therefore, $\langle G' \rangle \in HALF-CLIQUE$ as $G'$ is a graph with $2n-2k$ vertices.

Conversely,if $\langle G' \rangle \in HALF-CLIQUE$, that is $G'$ has a clique of size $n-k = k+(n-2k)$, then at most $n-2k$ of the clique come from the $n-2k$ new vertices. Therefore the reaming at least *k* vertices form a clique in *G*.

**Hence,** $\langle G,k \rangle \in CLIQUE$

When $k > \frac{n}{k}$ : if *G* has a $k-$clique, then $G''$ has a clique size $k = \frac{2k}{2}$ , and

Therefore, $\langle G' \rangle \in HALF-CLIQUE$ as $G''$ is a graph with $n+2k-n=2k$ vertices.

Conversely, if $\langle G'' \rangle \in HALF-CLIQUE$ , that is if $G''$ a clique of size has *k*, then the clique does not contain any of the new vertices as they are isolated.

**Thus, the clique is a *k* – clique of $G$ , and hence $\langle G,k \rangle \in CLIQUE$ .**

**Therefore,** the $HALF-CLIQUE$ **is *NP*- complete.**

Comment

## Problem

Let $CNF_k$ = { $\langle \phi \rangle \mid \phi$ is a satisfiable cnf-formula where each variable appears in at most k places}.

**a.** Show that $CNF_2$ ? P.

**b.** Show that $CNF_3$ is NP-complete.

## Step-by-step solution

**Step 1** of 4

**a)**

Consider the data:

$$CNF_k = \left\{ \begin{array}{l} \langle \phi \rangle \mid \phi \text{ is a satisfiable } cnf - \text{formula where each variable} \\ \text{appears in at most } k \text{ places} \end{array} \right\}$$

**CNF is Conjunctive normal form; it contains few rules.**

- A literal is Boolean variable or negated Boolean variable in the form
- Clause contains several literals connected with $\vee$s and $\wedge$s.

Comment

**Step 2** of 4

Now have to show that $CNF_2 \in P$.

<u>Class-P</u>: $P$ is a class of languages that are decidable in polynomial time on a deterministic single $-$tape Turing $-$machine.

Let $T_p$ be the polynomial time decider for $CNF_2$.

$T_p$ can be described as follows:

$T_p =$ " on input $\langle \phi \rangle$ ":

According to CNF rules, choose the clauses:

1. Consider the first clause of $\phi$. If it is of the form x, and there is a clause $\neg x$ in $\phi$, reject.
2. CNF is the form $x \vee A$, where $A$ is CNF. If x does not appear negated in other clauses, remove every clause of the form $x \vee B$ of $\phi$ and calculate the result $\phi$, if there is no clauses in $\phi$ then accept.
3. Solve CNF where $c$ occurs in every clause, where negation of $c$ does not appear.
4. When searching with $\phi$, if clauses found in the form $x \vee A$ and $\sim x \vee B$ then remove. Add $A \vee B$ in $\phi$
5. Go to step 1.

Every time $T_p$ processes each variable and reaches either accept or reject. Because of this the number of clauses in $\phi$ might decrease by 1 or 2. Hence running time of $T_p$ becomes polynomial time in terms of the number of variables.

So, $\boxed{CNF_2 \in P}$.

Comment

**Step 3** of 4

**b)**

Now have to show that $CNF_3$ is $NP$-complete.

$NP-$complete: A language $B$ is $NP-$complete if is satisfies two conditions:

1. $B$ is in $NP$
2. Every $A$ in $NP$ is polynomial time reducible to $B$.

**Step 1:** $CNF_3 \in NP$: If $CNF_3$ is in $NP$

- $V_p$ is a verified in polynomial time and it is described as follows:
  - $V_p =$ "on input $\langle\langle\phi\rangle, x\rangle$"

According to CNF rules, verify the following clauses:

- Verify each variable in $\phi$ which occurs in at most 3 places.
- Verify whether $x$ is a satisfying assignment in $\phi$.
- If both conditions are satisfied, then accept.
- Otherwise, reject.

Comment



**Step 2:** $3SAT \leq_p CNF_3$: It is best example for $CNF_3$ satisfying assignment.

Let $r_p$ be the polynomial time reduction from $3SAT$ to $CNF_3$.

When an input instance $\phi$ of $3SAT$, $r_p(\langle\phi\rangle)$ is given then construct an instance of $CNF_3$ from the following:

- First read from left to right, select the best example variable that access more than three times in the formula. Example variable as $S$ occurs in $m$ multiple places. $(x_1 \vee A_1), \ldots (x_m \vee A_m)$ where $x_i$ is $S$ or negated $S$.
- If nothing results more than three times, then output is $\phi$
- Select variables $S_1, \ldots, S_m$, If any $(x_i \vee A_i)$ remove from the formula
- $(S_1 \vee A_1) \wedge (\neg S_1 \vee S_2) \wedge (S_2 \vee A_2) \wedge (\neg S_2 \vee S_3) \ldots (S_m \vee A_m) \wedge (\neg S_m \vee S_m)$
- Go to step 1

Obviously reduced polynomial time $r_p(\langle\phi\rangle)$ is a formula identified that every variable occurs at most three times. It is also clear that $\phi$ is satisfiable if and only if $r_p(\langle\phi\rangle)$ is satisfiable. The $r_p$ is a reduced polynomial time in terms of the number of variable in $\phi$ from (1) and (2) $\boxed{CNF_3 \text{ is } NP\text{-complete}}$.

Comment

## Problem

Let $CNF_H$ = { $\langle \phi \rangle \mid \phi$ is a satisfiable cnf-formula where each clause contains any number of literals, but at most one negated literal}. Show that $CNF_H$ ? P.

## Step-by-step solution

**Step 1** of 5

$\mathrm{CNF_H} = \{<\varnothing>|$ is a satisfiable cnf-formula where each variable appears in at most k places$\}$ Hence, $CNF_H \in$ P can be proved by using following approach:

Situation is quite clear as all user needs to find the d**ependability** of $\mathrm{CNF}$ on $\mathrm{P}$ and $\mathrm{NP}$.

Comment

**Step 2** of 5

Consider the following algorithm which can be used for this language:

$N =$ "On input $\langle \varphi \rangle$ where $\varphi$ is a boolean formula in cnf"

1. If $\varphi$ don't consists a unit clause $(\sim x)$, assume every literals $x$ will be 1 and $(\sim x)$ will be 0 and accept.

2. Repetition performed until these exists no new $(\sim x)$ unit clause:

3. If $\varphi$ consist a unit clause $(\sim x)$, remove each clauses that contains $(\sim x)$ from $\varphi$ and remove every occurrences of $x$ from the clause in $\varphi$.

4. If an empty clause is exists in $\varphi$, reject.

5. Let every literals $x$ in $\varphi$ be 1, $(\sim x)$ be 0 and accept.

Comment

**Step 3** of 5

**Dependability can be viewed as**:

1. Consider that $\mathrm{CNF_2} \in \mathrm{P}$

Consider first situation of $\phi$ and consider it belongs to some x and if there is any $\neg x$ then reject. Consider one more situation of $\phi$ that will be belonging to some other variable let's say $\mathrm{y}$ and here situation will rejected if there is any $\neg y$ otherwise accepted.

• **Two situations are discussed here remove those situation from $\phi$ and relate those two situations, let's say M and N, to $\phi$ and call the result**. This way it is proved that $\mathbf{CNF_2} \in \mathbf{P}$.

Comment

**Step 4** of 5

2. Consider another situation $\mathrm{CNF_3} \in \mathrm{P}$

Consider first situation of $\phi$ and consider it belongs to some p and if there is any $\neg p$ then reject. Consider one more situation of $\phi$ that will be belonging to some other variable, let's say $\mathrm{q}$ and $\mathrm{r}$ here situation will rejected if there is any $\neg q$ otherwise accepted.

• **Two situations are discussed here remove those situation from $\phi$ and relate those two situations, let's say A and B, to $\phi$ and call the result.**

This v·^·γ it is proved that $\mathbf{CNF_3 \in P}$.

Comment

**Step 5** of 5

**Similarly, this situation will be held true for $\mathbf{CNF_H \in P}$.**

Comment

## Problem

Let ? be a 3cnf-formula. An ***≠-assignment*** to the variables of ? is one where each clause contains two literals with unequal truth values. In other words, an ≠ -assignment satisfies ? without assigning three true literals in any clause.

**a.** Show that the negation of any ≠ -assignment to ? is also an ≠ -assignment.

**b.** Let ≠ *SAT* be the collection of 3cnf-formulas that have an ≠ -assignment. Show that we obtain a polynomial time reduction from *3SAT* to ≠ *SAT* by replacing each clause $c_i$

$$(y_1 \vee y_2 \vee y_3)$$

with the two clauses

$$(y_1 \vee y_2 \vee z_i) \quad \text{and} \quad (\overline{z_i} \vee y_3 \vee b),$$

where $z_i$ is a new variable for each clause ci, and b is a single additional new variable.

**c.** Conclude that ≠ *SAT* is NP-complete.

## Step-by-step solution

**Step 1** of 4

**Assumption**: is a 3CNF formula.

$\phi$ is a clause that contains two literals with unequal truth values of an $\neq -$ assignment to the variable.

CNF is Conjunctive Normal Form. It has the following rules:

• A literal is Boolean variable or negated Boolean variable in the form.

• Boolean formula is in CNF called a CNF formula.

• If all clauses have three literals, then it is called 3cnf formula.

• Clause contains several literals connected with $\vee$s and $\wedge$s.

• Each clause has at least one satisfied literal and one unsatisfied literal in $\neq -$ assignment to $\phi$.

• The negation of an $\neq -$assignment conserve this property.

• Hence, negation of any $\neq -$to $\phi$ is also an $\neq -$assignment.

Comment

**Step 2** of 4

b)

**To show**: The formula $\phi$ is mapped to $\phi'$ then $\phi$ is satisfiable if $\phi'$ has an $\neq -$ assignment.

**Assumption:** $\neq$SAT is the collection of 3cnf formulas that have an $\neq -$assignment.

• Obtain a polynomial time reduction from 3SAT to $\neq -$SAT by replacing each close $c_i$ of the form $(y_1 \vee y_2 \vee y_3)$ with the two clause $(y_1 \vee y_2 \vee z_i)$ and $(\bar{z}_i \vee y_3 \vee b)$

Where

• $z_i$ is a new variable for each clause $c_i$

• b is a single additional new variable.

It is known that $SAT = \{(\phi) \mid \phi \text{ is a boolean formula}\}$

• Let $\phi$ and $\phi'$ are 3 CNF formulas of input and reduction on input $\phi$ and $\phi'$ as output.

• We must prove $\phi \in 3-SAT \Leftrightarrow \phi' \in \neq SAT$ therefore the reduction is correct.

Suppose that $\phi \in 3-SAT$

• $\phi$ is satisfiable.

• 1 is true and 0 is false. We get an $\neq$- assignment to $\phi'$ by extending a satisfying assignment to $\phi$ in such a way that we assign 1 to $\phi' \in k \neq SAT$.

• Else if both literals $y_1$ and $y_2$ are clauses $c_i$ are unsatisfied, else we assign 0 to $z_i$.

• Finally, we assign 0 to b.

• Extended assignment satisfies $\phi'$ and it is an $\neq -$ assignment to $\phi'$.

Therefore, $\boxed{\phi' \in \neq SAT}$

Comment

**Step 3** of 4

Suppose that $\phi' \in k \neq SAT$

• $\neq$ has an $\neq$- assignment

Satisfying assignment to $\phi$ as follows:

• From part(a) we obtain $\neq -$ assignment assigns 0 to b, otherwise simply negate the assignment.

• This $\neq -$ assignment cannot assign 0 to all $y_1, y_2$ and $y_3$ as doing so would force one of the two clauses, $(y_1 \vee y_2 \vee z_i)$ and $(\bar{z}_i \vee y_3 \vee b)$, to have all 0's

• Hence restricting this assignment to the variables of $\phi$ yields a satisfying assignment to $\phi$.

Therefore, 3SAT is polynomial time reducible to $\neq$SAT

Comment

**Step 4** of 4

(c)

**NP-COMPLETE:** A language B is NP-complete if it satisfies two conditions

• B is in NP

• Every A in NP is polynomial time reducible to B.

$\neq SAT \in NP$, as it is easy to verify whether an assignment is an $\neq -$ assignment.

Part(b) also proved $3SAT \leq_p \neq SAT$, $\neq$SAT is NP-complete.

Comment

## Problem

A ***cut*** in an undirected graph is a separation of the vertices V into two disjoint subsets S and T . The size of a cut is the number of edges that have one endpoint in S and the other in T . Let

$$MAX\text{-}CUT = \{\langle G, k\rangle \mid G \text{ has a cut of size } k \text{ or more}\}.$$

Show that *MAX-CUT* is NP-complete. You may assume the result of Problem 7.26. (Hint: Show that $\neq SAT \leq_P$ *MAX CUT*. The variable gadget for variable x is a collection of 3c nodes labeled with x and another 3c nodes labeled with $\overline{x}$. . The clause gadget is a triangle of three edges connecting three nodes labeled with the literals appearing in the clause. Do not use the same node in more than one clause gadget. Prove that this reduction works.)

## Step-by-step solution

**Step 1** of 2

***NP* – complete definition:**

A language *B* is *NP* – complete if it satisfies two conditions

1. *B* is in *NP*

2. Every *A* in *NP* is polynomial time reducible to *B.*

Comment

**Step 2** of 2

***MAX-CUT* is in *NP*.** We can guess the partition of the graph into two parts and verify that the number of edges cut is at least *k*.

We can show that *MAX-CUT* is in *NP* by showing that $\neq SAT \leq_P MAX - CUT$ :

• Let *n* be the number of variables and *c* be the number of clauses in the $\neq SAT$ instance $\phi$.

• We know that "$\neq SAT$ is the collection of $3cnf$ -formula that have an $\neq -$assignment" and

• "An $\neq -$assignment to a variable $\phi$ is one where each clause contains two literals with unequal truth values".

• Let *G* be the resulting graph.

• For every literal $z, G$ contain 2*c* nodes each labeled as *z* ( let's call this block" of nodes corresponding to *z*)

• Add all $(3c)^2$ edges between the block *z* and block $\overline{z}$ .

• For every clause, there is a triangle between three nodes that are labeled by the three literals that appear in that clause.

• Same node in a block cannot be used for more than one clause triangles.

• Now $G$ has $G$ nodes and $(3c)^2 n + 3c$ edges set $k = (3c)^2 n + 2c$

• We show that $\neq SAT$ has a $\neq -$assignment iff *G* has a cut of size at least *k.*

For forward direction,

assume that $a \neq -$assignment

• Place all nodes labeled by a TRUE literal on one side of the cut and all nodes labeled by a FALSE literal on the other side of the cut.

• This cuts all $(3c)^2 n$ edges between the blocks.

• Also Since every clause gets a TRUE and a FALSE literal, for every triangle. Two of the three edges are cut.

• Thus overall $(3c)^2 n + 2c$ edges are cut.

For the backward direction,

proof for any partition that cuts at least *k* edges must

1. Place every block on one side of the partition entirely.

2. Place blocks corresponding to complementary literals on opposite sides.

3. Therefore the partition defines an assignment to literals.

4. And then every clause must have a TRUE as well as a FALSE literal. So, two edges in that clause triangle get cut.

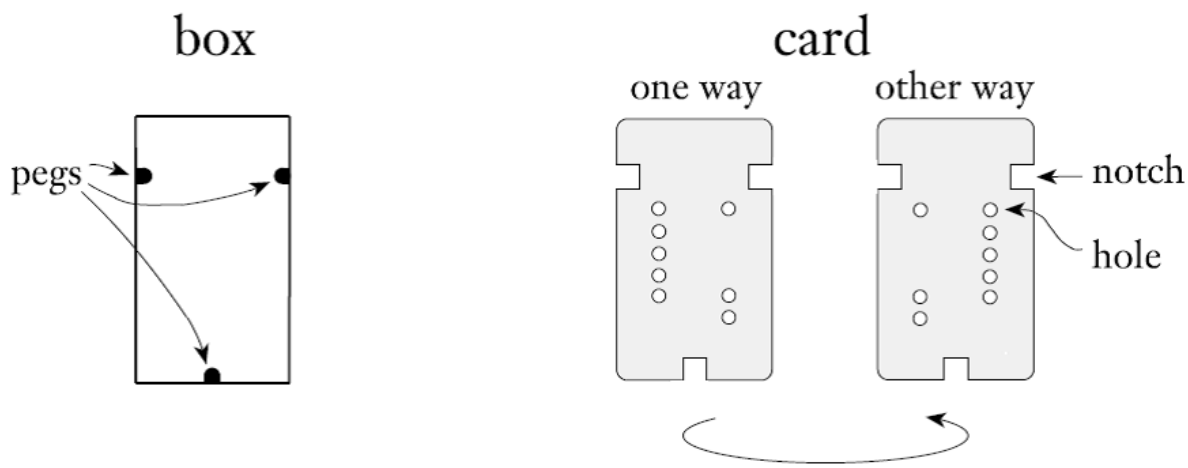
**Thus, $MAX-CUT$ is in *NP*–complete.**

Comment

## Problem

You are given a box and a collection of cards as indicated in the following figure. Because of the pegs in the box and the notches in the cards, each card will fit in the box in either of two ways. Each card contains two columns of holes, some of which may not be punched out. The puzzle is solved by placing all the cards in the box so as to completely cover the bottom of the box (i.e., every hole position is blocked by at least one card that has no hole there).

Let $PUZZLE = \{\langle c_1, \ldots, c_k \rangle \mid \text{ each } c_i$ represents a card and this collection of cards has a solution}. Show that *PUZZLE* is NP-complete.




## Step-by-step solution

**Step 1** of 4

A collection of cards with notches is to be fitted into a box with pegs. A card may fit into the box in one of two ways. Each card has two vertical columns of holes. If the cards are all placed so as to completely cover the bottom of the box the puzzle is solved.

Therefore, it can be prove that $PUZZLE$ is NP-complete where $PUZZLE$ is defined as:

$$PUZZLE = \{\langle c_1, \ldots, c_k \rangle \mid \text{ each } c_i \text{ represents a card and this collection has a solution}\}$$

• It can be observed that the order in which the cards are placed does not affect a solution to $PUZZLE$.

• First show the problem is in NP. Next NP-complete problems are proven to be polynomial time reducible to $PUZZLE$. This will prove that $PUZZLE$ is NP-complete.

Comment

**Step 2** of 4

A verifier $V$ for $PUZZLE$ will take the directions $d = d_1, \ldots, d_k$ in which the cards $c = c_1, \ldots, c_k$ are placed as the certificate.

The verifier $V$ is:

$V =$ "On input $\langle c, d \rangle$:

1. Place the cards $c$ in directions $d$ in the box.
2. Test if the bottom of the box is completely covered.
3. If yes, *accept*; else, *reject*."

• It can be seen that all the steps of the verifier take polynomial time because the placing of card in step 1 will take a polynomial time.

• Also, it will take a polynomial time to test whether the box is completely covered or not, because user have to traverse all the bottom of the box.

Hence this problem is NP.

Comment

**Step 3** of 4

To show that NP-complete problems are polynomial time reducible to $PUZZLE$.

For each and every instance of the $3$SAT problem for a Boolean formula $\phi$, it is shown how to construct an instance of the $PUZZLE$ problem.

• An instance $\phi$ of the $3$SAT problem for a Boolean formula $\phi$ with $l$ clauses and $3$ variables in a clause will be as follows.

• It can have $l \cdot 3$ variables, which are $x_1, \ldots, x_{3l}$.

$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_l \vee b_l \vee c_l)$, where each $a_j, b_j, c_j$ represents $x_i$ or $\overline{x_i}$.

• For reducing an instance of the $3$SAT problem into an instance of $PUZZLE$, the variable gadget and the clause gadget have to be created for $k$ cards with columns of $l$ holes on each card.

Comment

**Step 4** of 4

Looking at the condition for $PUZZLE$ to be satisfied, it is that all the punched out holes need to be covered. Thus, the clause gadget will be the condition for **a hole to be filled by a stack of cards.**

• Next, observe that a clause gadget is true when any of the three variables it contains are true.

• On this basis the variable gadget is made to be the placement of three cards chosen from the stack of cards such that the bottom of the box is not visible from a particular hole.

• This **reduction** $f$ **works in polynomial time for all instances of an NP-complete** problem, $3$SAT. The proven has also been proven to lie in NP.

**Therefore the problem** $PUZZLE$ **is NP-complete**

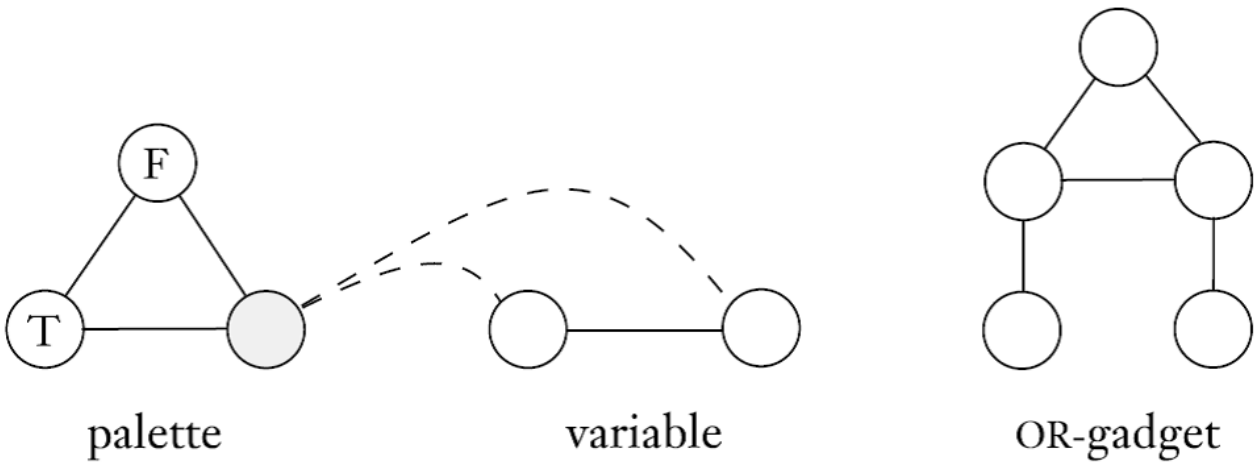
Comment

## Problem

A ***coloring*** of a graph is an assignment of colors to its nodes so that no two adjacent nodes are assigned the same color. Let

$$3COLOR = \{\langle G\rangle \mid G \text{ is colorable with 3 colors}\}.$$

Show that *3COLOR* is NP-complete. (Hint: Use the following three subgraphs.)




## Step-by-step solution

**Step 1** of 2

***NP* – complete:**

A language *B* is *NP* – complete if it satisfies following two conditions:

1. *B* is in *NP*

2. Every *A* in *NP* is polynomial time reducible to *B*.

Comment

**Step 2** of 2

1.3 COLOR is in *NP* because a coloring can be verified in polynomial time.

2. $3SAT \le_p 3COLOR$:

• " $3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable } 3cnf - \text{formula}\}$ " and "$3cnf$ -formula is the one in which all the clauses have three literals"

• Let $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_l$ be a $3cnf$ formula over the variable $x_1, \ldots, x_n$.

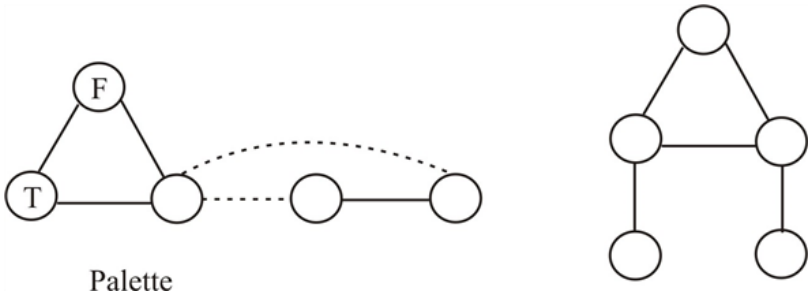
• To build a graph *G* with $2n + 6l + 3$ nodes, containing a variable gadget for each variable $x_i$, one clause gadget for each clause and one palette gadget as follows.

• Label the nodes of the palette gadget *T, F* and *R*.

• Label the node since each variable gadget + and – and cannot reach to the *R* node in the palette gadget.

• For each clause, create a gadget.

• Given three sub graphs.

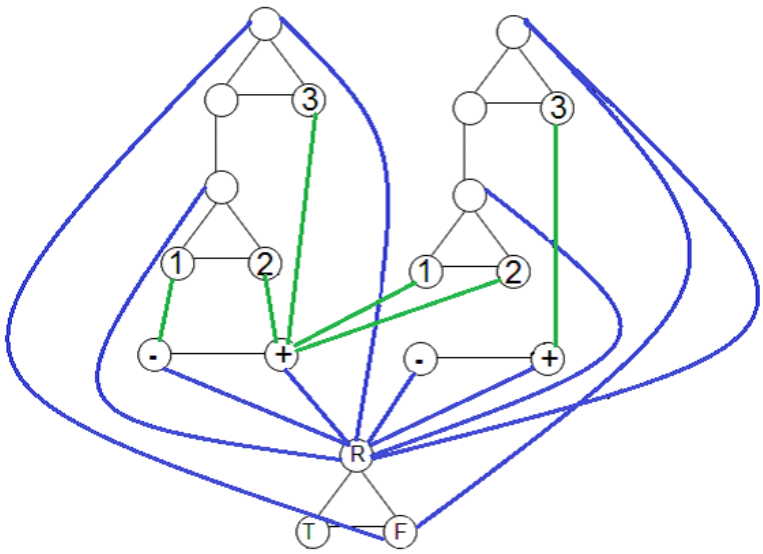



• Connect the *F* and *R* nodes to the top of the clause gadget in the palette.

• Also, connect the top of its bottom triangle to the *R* node.

• For every clause $c_j$, connect the $i^{th}\left(1 \le i \le 3\right)$ bottom node of its clause gadget to the literal node that appears in its $i^{th}$ location.

• An example is shown below.




• To show that the construction is correct, we first demonstrate that if $\phi$ is satisfiable, the graph is 3- colored.

• The three colors are called T, F and R.

• Color the palette with its labels.

• For each variable, color the + node T and – node F if the variable is true in a satisfying assignment: otherwise reverse the colors.

• Because each clause has one True literal in the assignment, we can color the nodes of that clause so that the node connected to the F node in the palette is not colored F.

• Hence we have proper 3-coloring.

• Similarly, if we are given a 3-coloring, we can obtain a satisfying assignment by taking the colors assigned to the + nodes of each variable.

• Observe that neither node of the variable gadget can be can be colored R, because all variable nodes are connected to the R node in the palette.

• Furthermore, if both bottom nodes of the clause gadget are colored F, the top node must be colored F, and hence, each clause must contain a true literal.

Hence, $3\text{SAT} \le_p 3$COLOR

**Therefore, from (1) and (2) 3 COLOR is NP- complete.**

Comment

## Problem

Let *SET-SPLITTING* = $\{\langle S, C\rangle \mid S \text{ is a finite set and } C = \{C_1, \ldots, C_k\}$ collection of subsets of S, for some k > 0, such that elements of S can be colored *red* or *blue* so that no $C_i$ has all its elements colored with the same color}. Show that *SET-SPLITTING* is NP-complete.

## Step-by-step solution

**Step 1** of 3

**NP –Complete:**

A language *B* is *NP*-complete if it satisfies 2 conditions

1. *B* is in *NP*

2. Every A in *NP* is polynomial time reducible to B.

Comment

**Step 2** of 3

1. **SET – SPLITING is in NP :**

SET – SPLITING is in NP because we can verify in polynomial time that no subset $C_i$ is monochromatic.

2. $3\,SAT \leq_p SET - SPLITING:$

To prove that the problem is NP complete, we give a polynomial time reduction from 3SAT to SET-SPLITING.

Given an instance of 3SAT $\phi$, set $S = \left\{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n, y\right\}$, where $x_i's$ are the variables and y is a special color variable.

Comment

**Step 3** of 3

**The splitting is done as follows:**

For every clause $C_i$ in $\phi$, Let $C_i$ be a subset of S containing the elements corresponding to the literally, in $C_i$ and the special elements $y \in s$, Then $C = C_1, \ldots, C_k$

If $\phi$ is satisfiable, consider a satisfying assignment.

If we color all the true literals red, all the false ones are blue, and y blue, then every subset $C_i$ of S has at least one red element (because it is satisfiable and it also contain one blue element y.

In addition, for a given splitting $\langle S, C\rangle$, we can able to set the literals that are colored differently from y to true.

In the same way, we can able to set the literals that have the same color as y to false.

This concludes that satisfying assignment for $\phi$.

**Thus, SET – SPLITTING is NP-Complete.**

Comment

## Problem

Consider the following scheduling problem. You are given a list of final exams $F_1, \ldots, F_k$ to be scheduled, and a list of students $S_1, \ldots, S_l$. Each student is taking some specified subset of these exams. You must schedule these exams into slots so that no student is required to take two exams in the same slot. The problem is to determine if such a schedule exists that uses only h slots. Formulate this problem as a language and show that this language is NP complete.

## Step-by-step solution

**Step 1** of 10

A list of final exams $F_1,\ldots,F_k$ is to be scheduled for a list of students $S_1,\ldots,S_l$, with each student taking a subset of the exams. The exams have to be scheduled into slots using only $h$ slots and so that no student has two exams in the same slot. This problem can be **formulated as a language** and also can be proved as "**it is NP-complete**".

Comment

**Step 2** of 10

This problem can be formulated as the following language $SCHEDULE$:

Comment

**Step 3** of 10

$$SCHEDULE = \left\{ \begin{array}{l} \langle F,S,h \rangle \mid F \text{ is the list of finals, } S \text{ is the list of subsets indicating which} \\ \text{finals each student is taking, and } h \text{ is the maximum number of slots the} \\ \text{schedule can take such that no student has more than one exam in a slot} \end{array} \right\}$$

Comment

**Step 4** of 10

The verifier $V$ for this language $SCHEDULE$ will take the schedule of time slots $T$ for the final exams $F$ as a certificate. It would then check each of the subsets of exams for the students, which are the members of $S$.

The **algorithm** for the verifier $V$ is:

$V =$ "On input $\langle F,S,h,T \rangle$:

1. For each sublist of $S$, test if more than one exam lies in the same slot of $T$.
2. If yes, *reject*.
3. Test if all elements of $F$ have a slot in $T$.
4. If no, *reject*.
5. Test if the number of slots used by $T$ is more than $h$.
6. If yes, *reject*. Else *accept*.

The steps 1, 3 and 5 of the verifier $V$ all take polynomial time so the verifier takes polynomial time. Therefore**, the language** $SCHEDULE$ **is in NP**.

Comment

**Step 5** of 10

In the $h$ Color Graph problem, the vertices of an undirected graph have to be colored in $h$ colors such that for each and every edge $(u,v)$, the vertices $u$ and $v$ are colored differently. Later it will be proven that the $h$ Color Graph problem is NP-complete.

• Now, try to reduce all instances of the $h$ Color Graph problem into instances of $SCHEDULE$ in polynomial time.

• An instance of the $h$ Color Graph problem with $k$ vertices can be mapped to an instance of $SCHEDULE$ with $k$ final exams and a maximum of $h$ slots taken by the schedule.

• The lists of exams taken by the students are a subset of the set of the $k$ vertices. An edge between two vertices indicates a conflict, which is at least one student needs to take both exams.

Comment

**Step 6** of 10

This reduction can be performed in polynomial time. It is valid as:

• If an instance of the $h$ Color Graph problem is true, then the instance of $SCHEDULE$ produced by the reduction will be true as exams (or vertices in the graph) colored identically can be scheduled in the same time slot and the length of the schedule will be $h$ time slots as there are $h$ different colors.

• From an instance of the $SCHEDULE$ problem, an instance of the $h$ Color Graph problem can be created. The vertices are created from the list of exams. Edges are drawn between vertices if at least a single student has to give both the related exams.

• The vertices corresponding to exams in the same time slot are colored identically. This instance of the Graph Color problem is satisfied as only the vertices that do not lie in the same time slot can have edge between them and such vertices shall not have the same color.

Comment

**Step 7** of 10

Now it needs to check which class of problems the $h$ Color Graph problem lies in. If it lies in NP-complete then the $SCHEDULE$ problem will also be in NP-complete as the NP-complete class of problems is reducible in polynomial time to it.

• The verifier $V$ for the $h$ Color Graph problem will take the $C=\{c_1,\ldots,c_k\}$ colors assigned to the $N=\{n_1,\ldots,n_k\}$ vertices as the certificate.

The **algorithm** for the verifier $V$ will be:

$V=$ "On input $\langle N,C\rangle$:

1. Test if all the colors $C$ are valid.
2. If no, *reject*.
3. For each vertex $n_i$ in $N$:
   Check if all adjacent vertices are colored differently.
4. If yes, *accept*; else *reject*.

The verifier takes polynomial time so the $h$ Color Graph problem is in NP. Following which, NP-complete problems have to be shown to be polynomial time reducible to the $h$ Color Graph problem**. The $h$ SAT problem is reduced to the $h$ Color Graph problem**.

Comment

**Step 8** of 10

An instance of the $h$ SAT problem will have $n$ clauses with each clause containing $h$ variables $v_1,\ldots,v_h$. The variable gadget will be the vertices of the graph labeled with the variables and the clause gadget will be the vertices (and their edges) added corresponding to them.

• An instance of the $h$ Color Graph problem is constructed by adding all the variables $v_1,\ldots,v_h$ and their complements $\overline{v_1},\ldots,\overline{v_h}$ as vertices of the graph.

•

Comment

**Step 9** of 10

All the variable vertices are connected to their complements by an edge. Label the pair formed by a variable vertex and its complement as $true$ for the variable and $false$ for the complement.

Comment

**Step 10** of 10

Next, add a new set of vertices $x_1, \ldots, x_h$ to the graph in the form of a clique (by joining each new vertex with all the other new vertices). Color these vertices $x_1, \ldots, x_h$ with colors $c_1, \ldots, c_h$.

• Also connect each $x_i$ with $v_j$ and $\overline{v_j}$ except for the case when $i = j$. Color one of $v_j$ or $\overline{v_j}$ with the same color as the corresponding $x_j$. This can be done with $h+1$ colors.

**Thus, the $h$ Color Graph problem has been proven to be NP-complete. Consequently the $SCHEDULE$ problem is also in NP-complete.**

Comment

## Problem

This problem is inspired by the single-player game *Minesweeper*, generalized to an arbitrary graph. Let G be an undirected graph, where each node either contains a single, hidden *mine* or is empty. The player chooses nodes, one by one. If the player chooses a node containing a mine, the player loses. If the player chooses an empty node, the player learns the number of neighboring nodes containing mines. (A neighboring node is one connected to the chosen node by an edge.) The player wins if and when all empty nodes have been so chosen.

In the *mine consistency problem*, you are given a graph G along with numbers labeling some of G's nodes. You must determine whether a placement of mines on the remaining nodes is possible, so that any node v that is labeled m has exactly m neighboring nodes containing mines. Formulate this problem as a language and show that it is NP complete.

## Step-by-step solution

**Step 1** of 5

Suppose that

$$MCP = \left\{ \begin{array}{l} \langle G, N \rangle \mid G \text{ is a graph } (V,E),\ N \text{ is labeling function } N : V \to \{X, 0, 1, 2, \ldots\} \\ \text{and there exists mine's placement on the set of nodes labeledso that every} \\ \text{numbers are consistence} \end{array} \right\}$$

The language $MCP$ is in NP, because in polynomial time one can easily test if a placement of mines is consistence. First users have to show $3SAT \leq_P MCP$, to prove that it is also NP-complete.

Comment

**Step 2** of 5

Consider $\varphi$ is a Boolean formula, now user want to convert it to $G$, which is an instance of the $MCP$ problem.

Here, suppose $c_1, c_{2,\ldots}$ are used to denote the clause appearing in $\varphi$. Now, suppose $x_1, x_{2,\ldots}$

Denotes the variable used in $\varphi$. Here, a variable $x_i$ 'appears' in $\varphi$ if one of $x_i$ or $\overline{x}_i$ appear in minimum single clause in $\varphi$.

For all the variables $x_i$ which appears in $\varphi$:

1. Three nodes are created as $x_i, x_i^f$ and $x_i^t$.

2. Edges are added $(x_i, x_i^t)$ and $(x_i, x_i^f)$, and

3. Now, set $N(x_i) = 1,\ N(x_i^t) = X$ and $N(x_i^f) = X$

Comment

**Step 3** of 5

For all the clause $c_i$ which appears in $\varphi$:

1. Three nodes are created as $c_i, c_i^1$ and $c_i^2$.

2. Edges are added from $c_i$ to the nodes corresponding to the three literals in $c_i$

3. Now, set $N(c_i) = 3,\ N(c_i^1) = X$ and $N(c_i^2) = X$

Comment

**Step 4** of 5

All instances of the $3$SAT problem can be reduced to an instance of the Circuit-SAT problem in polynomial time in a trivial manner by changing the Boole operators to a circuit of logic gates. The validity of this reduction needs to be proven.

• If there is an instance of the $3$SAT problem it can be converted to an instance of the Circuit-SAT problem by simply mapping Boolean operators to logic gates and connections between the gates.

• Setting the corresponding logic inputs to the Boolean values that satisfy the $3$SAT problem will satisfy this instance of the Circuit-SAT problem.

• Using an instance of the Circuit-SAT problem, an equivalent instance of the SAT problem is constructed by interchanging logic gates/wires with Boolean variables and operators. If the logical values that satisfy the instance of Circuit-SAT are changed into Boolean values, then the SAT instance will be satisfied.

Comment

**Step 5** of 5

**The Circuit-SAT problem is reducible in polynomial time to an NP-complete problem, so it is also in NP-complete. Hence, the $MCP$ problem is NP-complete.**

Comments (1)

## Problem

In the following solitaire game, you are given an m × m board. On each of its $m^2$ positions lies either a blue stone, a red stone, or nothing at all. You play by removing stones from the board until each column contains only stones of a single color and each row contains at least one stone. You win if you achieve this objective. Winning may or may not be possible, depending upon the initial configuration. Let *SOLITAIRE* = { $\langle G \rangle$ I G is a winnable game configuration}. Prove that *SOLITAIRE* is NP-complete.

## Step-by-step solution

**Step 1** of 3

**Definition of NP- Complete:**

A language B is NP- Complete if it satisfies two conditions.

1. B is in NP

2. Every A is NP is polynomial time reducible to B i.e. B is NP-hard.

**Objective of the SOLITAIRE Game:**

• SOLITAIRE Game requires a $m \times m$ board.

• On each $n^2$ positions of $m \times m$ board a blue stone or a red stone or nothing is placed.

• Now the game is to remove the stones so that each column contains only stones of single color and each row contains at least one stone.

• The people who achieve this objective will win the game.

Now we have to show that SOLITAIRE is NP-Complete. Before this, we have to show that SOLITAIRE is in NP.

Comment

**Step 2** of 3

**SOLITAIRE is in NP:**

SOLITAIRE $\in NP$ because it can be verified that a solution works in polynomial time.

Every Language in NP is polynomial time reducible to SOLITAIRE:

• We know that $\text{"3SAT} = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3 cnf - formula}\}\text{"}$, three variables.

And $\text{"3SAT is NP- complete"}$

So, if we show that $\text{3SAT} \leq_P$ SOLITAIRE then SOLITAIRE is also NP- Complete.

• Given $\phi$ with m variables $V_1, \ldots, V_m$ and k clauses $C_1, \ldots, C_k$

• Now construct the following game g with $k \times m$ board.

**Construction of $k \times m$ game of G:**

Let us assume that $\phi$ has no clauses that contain both $V_i$ *and* $\overline{V_i}$ because such clauses may

If the variable $V_i$ is in clause $C_i$ then put a blue stone in row $C_i$ column $V_i$

If the variable $\overline{V_i}$ is in clause $C_i$ then put a red stone in row $C_i$, column $V_i$ then

$k \times m$ board can be changed to square board necessary without affecting solvability.

Comment

**Step 3** of 3

**Now we need to show that $\phi$ is satisfiable if and only if G has a solution:**

If $\phi$ is satisfiable then G has a solution(Forward direction):

- A Satisfying assignment is taken.
- If $V_i$ is true, remove the rod stone from the corresponding column.
- If $V_i$ is false, remove the blue stone from corresponding column.
- So, stones corresponding to true literals remains.
- Because every clause has a true literal, every row has a stone.
- Therefore G has a solute or.

If G has a solution then $\phi$ is satisfiable (*backward direction*):

- Take a game solution.
- If the red stone removed from a column, set the corresponding variable true.
- If the bluestone is removed from a column, set the corresponding variable false.
- Every row has a stone remaining, so every clause has a true literal.
- Therefore $\phi$ is satisfied

**Thus, SOLITAIRE is NP-Complete.**

Comment

## Problem

Recall, in our discussion of the Church–Turing thesis, that we introduced the language D = { $\langle p \rangle$ | p is a polynomial in several variables having an integral root}. We stated, but didn't prove, that D is undecidable. In this problem, you are to prove a different property of D—namely, that D is NP-hard. A problem is ***NP-hard*** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself. So you must show that all problems in NP are polynomial time reducible to D.

## Step-by-step solution

**Step 1** of 3

Consider a language $D$ (as discussed in **Church-Turing thesis**), which is defined as:

$$D = \{\langle p \rangle \mid p \text{ is a polynomial in several variable having an integral root}\}$$

Here, the given language $D$ can be shown as $NP-hard$. This can be obtained by using the fact of $NP-completeness$ property of the language $D$.

Comment

**Step 2** of 3

Now, consider the way to proof the $NP-completeness$ property of the given language. The language $D$ can be shown as $NP-complete$ by using a reduction from $3SAT$.

• An expression $L$ in $3SAT$ can be converted into $L'$ by using the following facts:

$\sim a = (1-a)$ (Where the sign (~) is defined as a **NOT logic** operator)

$a \text{ and } b = (ab)$

$a \text{ or } b = (a+b-ab)$

• Now, for simplicity, a function $f(a,b,c)$ could be defined for each of the eight possibilities for all the clauses in $3SAT$. Here, eight possibilities will be taken because each variable $a, b \, and \, c$ will acquire a value $1/0$.

Comment

**Step 3** of 3

An algorithm can be used which replace every clause with one of the replacements and also used for putting the symbol multiplication between them. In other word, the description of above algorithm can be given as:

$$f(a \text{ or } b \text{ or} \sim c) = \left[(a+b-ab)+(1-c)-(a+b-ab)(1-c)\right]$$

**The running time of the above given algorithm is linear**.

• The result $L'$ shows a polynomial behavior that consist **an integral root if and only if** $L$ **is in** $SAT^*$. Therefore, the language $D$ can be said as $NP-complete$.

• It is already known to be **un-decidable**, thus it will **not exist in** $NP$**.** Therefore from the above explanation, it can be said that **the language** $D$ **is** $NP-hard$.

Comment

## Problem

A subset of the nodes of a graph G is a ***dominating set*** if every other node of G is adjacent to some node in the subset. Let

$$DOMINATING\text{-}SET = \{\langle G, k\rangle \mid G \text{ has a dominating set with } k \text{ nodes}\}.$$

Show that it is NP-complete by giving a reduction from *VERTEX-COVER*.

## Step-by-step solution

**Step 1** of 3

**NP-Complete:**

A language *B* is NP-Complete if it satisfies 2 conditions.

1. *B* is in NP

2. Every *A* in NP is polynomial time reducible to *B*.

Comment

**Step 2** of 3

**1. *DOMINATING – SET* is in NP:**

• Consider an instance $\langle G,k\rangle$ of the *DOMINATING-SET* and a covering *D*.

• Check that each node of *G* is adjacent to some node in *D*.

• This can be done in polynomial time.

• Therefore, *DOMINATING-SET* is in NP.

Comment

**Step 3** of 3

**2.** $\textbf{\textit{VERTEX - COVER}} \leq_P \textbf{\textit{DOMINATING - SET}}$

Now show that *VERTEX-COVER* reduces to *DOMINATING-SET*.

• Consider an instance $\langle (V,E),k\rangle$ of *VERTEX-COVER*.

• Construct an instance $\langle ((V-S)\cup V', E\cup E'),k\rangle$ of *DOMINATING-SET* where $S\subseteq V$ are nodes of degree 0.

• For each edge $(u,v)\in E$, there are edges $(u,w)$ and $(w,v)$ in $E'$ where $w\in v'$ in new vertex corresponding to $(u,v)$.

• Let $G=\langle V,E\rangle$ and $G'=\langle (V-S)\cup V', E\cup E'\rangle$.

• Suppose $(\langle V,E\rangle,k)$ is in *VERTEX-COVER*.

• There exits $C\subseteq V$ of size *k* where each edge $(u,v)\in E$ has either $u\in C$ or $v\in C$.

• If $v\in(V-S)$ then the degree of $v$ is one or more then there exists a node $u$ such that $(u,v)\in E$ which implies that at least one of *u* or *v* is in *C*. Thus, $v$ is covered.

• If $w\in V'$ then *w* is adjacent to both *u* and *v* where $(u,v)\in E$ which implies that at least one of *u* or *v* is in *C*. Thus, *w* is covered.

In other direction, suppose that $\langle ((V-S)\cup V', E\cup E'),k\rangle$ is in *DOMINATING-SET*.

• Then there exists $C\subseteq((V-S)\cup V')$ of size $k$.

• In such cases in which multiple such $C$ exist, it can be said that at least one includes no vertices in $V'$.

• This is always exists since $w \in (C \cap V')$ that corresponds to edge $(u,v)$ covers only nodes $u, v, w,$ but using $u$ instead of $w$ covers $u, v, w,$ and possibly more.

• Therefore, $C \subseteq (V - S)$ and $C$ is a vertex cover for $G$.

• This is because $C$ is a *DOMINATING-SET* for $G'$ implying that all nodes of $V'$ are covered. Thus, every edge $(u,v) \in E$ has at least one of $u, v$ in $c$.

Therefore, $VERTEX\text{-}COVER \leq_p DOMINATING\text{-}SET$.

**From (1) and (2), *DOMINATING-SET* is NP-complete.**

Comment

## Problem

Show that the following problem is NP-complete. You are given a set of states Q = {$q_0$, $q_1$, . . . , $q_l$} and a collection of pairs {($s_1$, $r_1$), . . . , ($s_k$, $r_k$)} where the $s_i$ are distinct strings over ∑ = {0, 1}, and the $r_i$ are (not necessarily distinct) members of Q. Determine whether a DFA M = (Q,∑, ?, $q_0$, F) exists where ?($q_0$, $s_i$) = $r_i$ for each i. Here, ?(q, s) is the state that M enters after reading s, starting at state q.

(Note that F is irrelevant here.)

## Step-by-step solution

**Step 1** of 2

Correct DFA which satisfy $C$ constraints and in polynomial time $\Pi$ can be guessed by Non Deterministic Turing Machine iff such DFA available or exist.

For showing that problem is NP complete reduce it to polynomial time.

Consider the formula $F = \wedge_{j=1}^{m} R_j$ where $R_j = \left(s_j \vee t_j \vee u_j\right)$ and construction some constraints $C$ and $\Pi$.

- $C = \{c_T, c_F, c_1, c_2\}$ are states.
- Creating pair $(\varepsilon, c_F)$ in $\Pi$ for enforcing $c_F$ as starting state.
- Every variable $s$ belongs to $F$ will create the pairs $(s\bar{s}, c_T)$ and $(\bar{s}s, c_T)$.
- Every clause $R_j{}'$ in formula $F$ will have pair in $\Pi$ that is $(s\#_s, c_1)$ and $(\bar{s}\#_s, c_2)$ that enforces that when reading $s$ and $\bar{s}$, $DFA$ must be in different state.
- Choose any $s$ in $F$. Now for $\forall$ variable $t$ create other three points in $\Pi$: $(s\bar{s}t, c_T)$, $(s\#_s t, c_1)$, $(\bar{s}\#_s t, c_2)$.

Comment

**Step 2** of 2

$F$ is satisfiable iff there is some $DFA$ that satisfy $C$ and $R$. Reduction is taking some polynomial time therefore given problem is NP-complete.

Comment

## Problem

Let $U = \{\langle M, x, \#^t \rangle |$ NTM M accepts x within t steps on at least one branch}. Note that M isn't required to halt on all branches. Show that U is NP-complete.

## Step-by-step solution

**Step 1** of 2

User contains an NDTM $M_L$, for any given NP language, in such a way that $\forall x \in L$, $M_L$ accepts $x$ on minimum single branch in maximum $p_L(|x|)$ steps. Here, $p_L(\ )$ denotes a fixed polynomial depending on the machine.

• It is also known that any $x \notin L$ will not be accepted by $M_L$. Then, user can use a polynomial time to create $y = \langle M_L, x, \#^{p_L(|x|)} \rangle$ for the given $x$.

• Consider the previous argument, according to this argument, $x \in L$ if and only if $y \in U$. Therefore, ***U* is NP-hard**.

Comments (1)

**Step 2** of 2

To show that *U* is also in NP, users have to create an NDTM $M_U$, which given an input $y = \langle M_L, x, \#^t \rangle$, simulates *M* on *x* for *t* steps.

• Every branches of *M* are guessed by $M_U$ non-deterministically and accepts *u* if and only if *u* is accepted by *M.*

• Since the minimum length of input is *t* and user simulate *M* for maximum *t* steps, the running time is polynomial in the length of the input.

• Now it can be easily seen that the language *U* is exactly accepted by $M_U$, therefore, $U \in NP$.

Hence**, *U* is NP-complete.**

Comment

## Problem

Show that if P = NP, a polynomial time algorithm exists that produces a satisfying assignment when given a satisfiable Boolean formula. (Note: The algorithm you are asked to provide computes a function; but NP contains languages, not functions. The P = NP assumption implies that *SAT* is in P, so testing satisfiability is solvable in polynomial time. But the assumption doesn't say how this test is done, and the test may not reveal satisfying assignments. You must show that you can find them anyway. Hint: Use the satisfiability tester repeatedly to find the assignment bit-by-bit.)

## Step-by-step solution

**Step 1** of 2

• SAT or SATISFIABILTY problem probes whether there exists an interpretation of the fed-in input that satisfies (or returns TRUE) a given Boolean formula.

• $\mathrm{P=NP}$ implies the existence of a polynomial time algorithm to test the decidability of SAT. By decidable, we mean that a Turing Machine (TM) exists for our SAT problem that either outputs ACCEPT or REJECT.

• Consider the below described algorithm:

For a TM U:

$U$ = "on input $\alpha$ , where $\alpha$ is a Boolean formula of variables $x_1, x_2, \ldots, x_n$".

1. Run $\mathrm{U}$ on $\alpha$. If our Boolean input $\alpha$ is not satisfiable, reject. This means that no matter what interpretations we might try to achieve for input $\alpha$, our TM would never return TRUE.

2. For $i$ from $1 \text{ to } \mathrm{n}$:

3. Replace all the $\mathrm{x_i s}$ in $\alpha$ with 1 and simulate our TM U on that. For example: say, for $\alpha = 0_{x_1} 1_{x_2} 1_{x_3} 0_{x_4} 1_{x_5} 1_{x_6}$, we shall first replace $x_1(0)$ with 1 & then in the next iteration $x_2$ with 1, & so on.

4. If $U$ accepts, we perform a permanent overwrite for $x_i$ with 1, otherwise write $\mathrm{x_i}$ as 0.

This algorithm is deterministically in P, since both the "for loop" & "replacement of $\alpha$ in for loop" have polynomial running times.

Comment

**Step 2** of 2

**Since,** $\text{polynomial*polynomial} = \text{polynomial}$ **devised algorithm is polynomial time.**

Comment

## Problem

Show that if P = NP, you can factor integers in polynomial time. (See the note in Problem 7.38.)

## Step-by-step solution

**Step 1** of 1

An integer can be factorized in multinomial time if $P = NP$ has given.

• Consider the language

$L = \{\langle n, x, y\rangle \mid n \text{ has a factor } p \text{ in the range } x \le p \le y\}$.

• Assume $P = NP$ and the above language can be decided by a polynomial algorithm.

• Each time the search space is divided into half by using repeated applications of the algorithm.

• The repetition is done by asking "Is a factor exists in the range $\left(x, \frac{x+y}{2}\right)$?". If the factor is not in this region then it can be said that there is a factor in other range.

• The number of times the algorithm applied here is sufficient to $\log n$. In other words it can also be said that, it will take $O(k)$, if $k$ is the number of bits exists in $n$. So, one factor can be isolated by using the algorithm which consist polynomial number of applications.

**Since it has maximum $O(k)$ factors, therefore it can be said that, every factors can be found in polynomial time.**

Comment

## Problem

Show that if P = NP, a polynomial time algorithm exists that takes an undirected graph as input and finds a largest clique contained in that graph. (See the note in Problem 7.38.)

## Step-by-step solution

**Step 1** of 2

**Class - *P* :** $P$ is a class of Languages that are decidable in polynomial time on a deterministic single tape Turing machine.

**Class - *NP*:** $NP$ is a class of Languages that are decidable in polynomial time on a nondeterministic Turing machine.

**Clique:** A clique in an undirected graph is a sub graph, wherein every two nodes are connected by an edge.

• If $P = NP$ then we have to show that "a polynomial time algorithm exists that takes an undirected graph as input and finds the largest clique in the graph.

• A k – clique is a clique that have k-nodes.

• We know that "clique is in $NP$".

• So if $P = NP$ then clique is in $P$.

**Therefore, if $P = NP$ then clique is recognizable in polynomial time.**

Comment

**Step 2** of 2

The following algorithm will find the largest clique in the graph:

1. Let $n$ be the no. of nodes in the given graph $G$.

$i$ be the variable which runs from 1 ton.

2. Using the polynomial time algorithm form clique, check whether there exist a clique of size $i$.

3. Output the Large $\mathrm{i}$ for which a clique exists.

To find the maximum clique, we start with $i$, the maximum clique size.

Remove one node and see if there is still a clique of size $i$.

If not, restore that node and remove another node.

If so, respect the process until we are left with a graph of $\mathrm{i}$ nodes, which must be a clique.

• This algorithm will take almost $\mathrm{n}$ trials to find which node to remove and at most $\mathrm{n}$ nodes to be removed.

**Then the total running time is polynomial.**

Comment

## Problem

In the proof of the Cook–Levin theorem, a window is a 2 × 3 rectangle of cells. Show why the proof would have failed if we had used 2 × 2 windows instead.

## Step-by-step solution

**Step 1** of 1

**Cook – Levin theorem:**

$SAT$ is $NP-\text{Complete}$ .This theorem restates that $SAT \in P$ iff $P = NP$.

• In the proof of the Cook-Levin theorem, a window size to be a $2\times 3$ rectangle of cells.

• If we had used $2\times 2$ windows that we can only use $2\times 2$ sub windows of the ones, obtained by deleting the leftmost or the rightmost column.

• Legal window $2\times 3$ is as follows.

| | | |
|---|---|---|
| $a$ | $q_1$ | $b$ |
| $q_2$ | $a$ | $c$ |

• In the window of figure (head move left), the right two columns allow the head to move left.

• The left two columns allow the head to move left into a state $q_2$, but this state is no longer restricted by what symbol was scanned by the head.

• So if there is some state $q_2$ into which it is possible to move while moving left, this window allows switching into this state on any left move.

• This will allow typically many tableaux (and so many satisfying assignments) that do not correspond to computations of our nondeterministic Turing machine.

Comment

## Problem

Consider the algorithm *MINIMIZE*, which takes a DFA M as input and outputs DFA M'.

$MINIMIZE =$ "On input $\langle M \rangle$, where $M = (Q, \Sigma, \delta, q_0, A)$ is a DFA:

1. Remove all states of $M$ that are unreachable from the start state.
2. Construct the following undirected graph $G$ whose nodes are the states of $M$.
3. Place an edge in $G$ connecting every accept state with every nonaccept state. Add additional edges as follows.
4. Repeat until no new edges are added to $G$:
5. For every pair of distinct states $q$ and $r$ of $M$ and every $a \in \Sigma$:
6. Add the edge $(q, r)$ to $G$ if $(\delta(q, a), \delta(r, a))$ is an edge of $G$.
7. For each state $q$, let $[q]$ be the collection of states $[q] = \{r \in Q |$ no edge joins $q$ and $r$ in $G\}$.
8. Form a new DFA $M' = (Q', \Sigma, \delta', q_0{}', A')$ where
   $Q' = \{[q] | \, q \in Q\}$ (if $[q] = [r]$, only one of them is in $Q'$),
   $\delta'([q], a) = [\delta(q, a)]$ for every $q \in Q$ and $a \in \Sigma$,
   $q_0{}' = [q_0]$, and
   $A' = \{[q] | \, q \in A\}$.
9. Output $\langle M' \rangle$."

**a.** Show that $M$ and $M'$ are equivalent.

**b.** Show that $M'$ is minimal—that is, no DFA with fewer states recognizes the same language. You may use the result of Problem 1.52 without proof.

**c.** Show that *MINIMIZE* operates in polynomial time.

## Step-by-step solution

**Step 1** of 3

• Consider if $M$ accepts the string $s = s_1 s_2 s_3 \ldots s_l$ of the length $l$ then there must be some states in sequence $\left\{ \begin{array}{l} (q_{j_0}, q_{j_1}, q_{j_2} \ldots\ldots, q_{j_l}) : q_{j_0} = q_0 \\ \text{and } q_{j_j} = \delta(q_{j_{i-1}}, s_i) \end{array} \right\}$ and $[q_{j_l}] \in F$. That shows $M'$ accept $s$ depends on states sequence $\left\{ \begin{array}{l} [q_{j_0}], [q_{j_1}], [q_{j_2}] \ldots\ldots, [q_{j_l}] : [q_{j_0}] = q_0 \\ \text{and } q_{j_i} = \delta'(q_{j_{i-1}}, s_i) \end{array} \right\}$ and $[q_{j_l}] \in F'$.

Hence, $L(M)$ is subset of $L(M')$.

• Now if some other string $u = u_1 u_2 u_3 \ldots u_l$ of $l$ length accepted with $M'$, consider $\{[q_{j_0}], [q_{j_1}], [q_{j_2}] \ldots\ldots, [q_{j_l}] : [q_{j_0}] = q_0 \text{ and } q_{j_i} = \delta'(q_{j_{i-1}}, s_i)\}$ and $[q_{j_l}] \in F'$. By induction when $u$ is input to $M$, now the corresponding sequence state which are visited by $M$, say $p_0, p_1, \ldots, p_l$ such that $p_0 = q_0$ and $r_i \in |q_{j_i}| \forall i$.

As $[q_{j_l}] \in F'$ and $[q_{j_l}] \in F$, hence $p_l \in F$ therefore $L(M') \subseteq L(M)$

Comment

**Step 2** of 3

• Consider $\delta(q_0,s)$ is state of $M$ after reading $s$ after start from $q_0$. Two different states $p$ and $q$ in graph which is undirected is connected by edge iff there exist $s$ and $u$ strings. Such that $\delta(q_0,s)=q$ and $\delta(q_0,u)=p$.

Based on given case $|q|$ will store states $q'$ so that $s$ and $u$ with $\delta(q_0,s)=q$ and $\delta(q_0,u)=q'$ such that $s$ and $u$ indistinguishable. For $q' \in [q]$ hence $q' = [q]$

• As statement given by theorem Myhill Nerode Deterministic Finite Automata recognize $L(M)$ must be $|Q'|$ number of states.

Hence, $M'$ also have $|Q'|$ number of states and $L(M)$ will be equal to $L(M')$ and $M'$ will minimal.

Comment

**Step 3** of 3

• Consider $|Q| = n_1$. In given algorithm Step 1 will take $O(n_1^2|\Sigma|+n_1^3)$, by Brute force algorithm. Also 3rd 5th and 6th Step will take $O(n_1^2)$ time and each repetition will take $O(n_1^2\Sigma)$ time.

• 10th Step will complete in $O(n_1^3)$. In 8th step it will check that either $|q|=|r|$ or not that takes $O(n_1^2)$

• When construct final Deterministic Finite Automata $M'$, will additional take additional $O(n_1^2\Sigma)$ time.

Comment

## Problem

For a cnf-formula ? with m variables and c clauses, show that you can construct in polynomial time an NFA with $O_{(cm)}$ states that accepts all nonsatisfying assignments, represented as Boolean strings of length m. Conclude that P ≠ NP implies that NFAs cannot be minimized in polynomial time.

## Step-by-step solution

**Step 1** of 2

Consider a cnf-formula $\varnothing$, which consist $c$ clauses and $m$ variables. A $NFA$, $D$ is constructed in such a way that, on input $\varnothing$, one of the $c$ clauses is picked deterministically. Also, it reads an input of length $m$ and checks it for acceptance. In other words, it accepts it, if the clause is not satisfied by it and otherwise reject. In addition, all the inputs are accepted by this $NFA$ $D$ which is also not equal to $m$.

• There is a need of $O(m)$ states for every clause. So that, $D$ consist $O(cm)$ states. Therefore, it is obvious that a polynomial time can be used to compute $D$.

• Now, minimum one of the clause is not satisfied for any non-satisfying assignment $q$. So in this case $q$ is accepted by $D$.

• Now, by using the inverse of the above statement, if $q$ is accepted by $D$ then some clause are not satisfied.

**Hence**, from the above discussion it can be said that, **every non-satisfying assignments of $\varnothing$ is accepted by $D$.**

Comment

**Step 2** of 2

Now, suppose a polynomial time can be used for minimizing $NFA$ problem. Consider a polynomial time algorithm with a cnf-formula $\varnothing$, which consist $c$ clauses and $m$ variables.

• Now, the construction of a $NFA$ $D$ takes place in such a way that it accepts every non-satisfying assignments of $\varnothing$.

• It can be noticed from the above statement that **every binary string will be accepted by $D$,** if and only if $\varnothing$ **is not satisfiable**.

• Now, a new $NFA$ $D'$ is generated by running the $NFA$ minimizing algorithm. If $D'$ consists only a **single state**, which accepts every binary strings then **reject** $\varnothing$, otherwise $\varnothing$ is **accepted**.

So, **an algorithm in polynomial time for $3SAT$ can be obtained**. Hence, it can be said that $P = NP$. In other way $P \neq NP$ **implies that $NFA$ cannot be determined in polynomial time.**

Comment

## Problem

A ***2cnf-formula*** is an AND of clauses, where each clause is an OR of at most two literals. Let

Let $2SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 2cnf-formula}\}$. Show that $2SAT \in \mathrm{P}$.

## Step-by-step solution

**Step 1** of 5

Class – P: $\mathrm{P}$ is a class of Languages that are decidable in polynomial time on a deterministic single – tape Turing – machine.

The Language is $2\mathrm{SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable } 2\,\mathrm{CNF} \text{ formula}\}$

A $cnf-\text{formula}$ is said to be $2\ cnf$ if all the clauses have two literals.

Now we have to prove that $2\,SAT \in P$.

• Let $\phi$ be the $2\,cnf-\text{formula}$ one variables $x_1, x_2, \ldots, x_n$

• Let us construct the graph $G$ for the give $\phi$ as follows :

→ The variables and their negations in $\phi$ are taken as vertices of graph $\mathrm{G}$. That is, $V = \{x_1, \ldots, x_n\} \cup \{\bar{x}_1, \ldots, \bar{x}_n\}$.

→For every clause of the form $\mathrm{A} \vee \mathrm{B}$ in $\phi$, add a directed edge from $\bar{\mathrm{A}}$ to $\mathrm{B}$ and one from $\bar{\mathrm{B}}$ to $\mathrm{A}$ in graph $\mathrm{G}$.

• So by the construction of the graph, it is follows that, if there is an edge from *A* to *B* then there is an edge from $\bar{\mathrm{B}}$ to $\bar{\mathrm{A}}$.

• Now let us suppose that there is a directed path from $x_i\ to\ \bar{x}_i$ and from $\bar{x}_i\ to\ x_i$.

• The existence of a directed path from $x_i\ to\ \bar{x}_i$ is equivalent to saying that $x_i \Rightarrow \bar{x}_i$ and the existence of a directed path from $\bar{x}_i\ to\ x_i$ is equivalent to saying that $\bar{x}_i \Rightarrow x_i$. Together, they implying that $x_i \cong \bar{x}_i$, which is false.

• So if this condition occurs then the formula $\phi$ has an un-satisfiable clause embedded in it.

• Conversely we will show that, if there is no such pair of paths (one form $x_i\ to\ \bar{x}_i$ and another from $x_i\ to\ \bar{x}_i$) then a satisfying assignment can be found for $\phi$, by the following algorithm.

Comment

**Step 2** of 5

Step – 1 For each variable $x_i$, check if there is a path from $x_i\ to\ \bar{x}_i$. If there is such a path, assign $x_i = false$. For all variables $V$ such that there is a path from $\bar{x}_i\ to\ v$, assign $v$ to true and $\bar{v}$ to false.

Comment

**Step 3** of 5

Step – 2 for each variable $x_i$, check if there is a path from $\bar{x}_i\ to\ x_i$. If there is such a path, assign $x_i = true$. For all literal $v$ such that there: a path form $x_i\ to\ v$, assign $v$ to true and $\bar{v}$ to false.

Comment

**Step 4** of 5

Step – 3 Propagate all the “true values” down the paths, and the “false values” up the paths.

Comment

**Step 5** of 5

- So this algorithm never assigns both true and false values to the same variable.
- The entire algorithm will be executed in polynomial time.
- Therefore $2SAT$ is in $P$.

Comment

## Problem

Modify the algorithm for context-free language recognition in the proof of Theorem 7.16 to give a polynomial time algorithm that produces a parse tree for a string, given the string and a CFG, if that grammar generates the string.

THEOREM **7.16**

Every context-free language is a member of P.

## Step-by-step solution

**Step 1** of 3

Parse tree is a tree of any grammar $G=(V,T,R,S)$ where start symbol *S* is root, non-terminals are interior nodes and terminals are leaf nodes. Each terminal string which is generated by some grammar *G* has parse tree.

Comment

**Step 2** of 3

Here, consider the CFG *G* where,

- *V* is variables set.
- *T* is terminals or leaf nodes.
- *R* is rule of production.
- *S* is starting symbol.

Comment

**Step 3** of 3

The Theorem 7.16 states that every context free language is a member of P. There is an algorithm for the Theorem 7.16. Modify the algorithm to produce a polynomial time algorithm that produces a parse tree for a string using the CFG.

The algorithm is as follows:

```
D = "On input s = s1 s2 ....sn and CFG G
    ParseTree ← (Initial S tree, Beginning of the string s)
    CurrentState ← POP the ParseTree
    repeat for i = 1,2,3,4....n
        if parsing is successful for CurrentState
            return TREE(CurrentState)
        else
            //Expand the CurrentState
            if no nodes to expand
                return reject
            else
                //Apply the rules to find the next node
                temp ← Apply the CFG rules on CurrentState
                PUSH temp into the ParseTree
        if ParseTree is empty
            return reject
        else
            // Move to the next part of the input
            CurrentState ← NEXT(ParseTree)
    return ParseTree
```

The above algorithm takes the string and apply the rules on it. For each part of the string *s*, apply the rules and find the next node (i.e., state). Each new state is added to the tree. If the input is processed completely then it returns the parse tree.

## Problem

Say that two Boolean formulas are ***equivalent*** if they have the same set of variables and are true on the same set of assignments to those variables (i.e., they describe the same Boolean function). A Boolean formula is ***minimal*** if no shorter Boolean formula is equivalent to it. Let *MIN-FORMULA* be the collection of minimal Boolean formulas. Show that if P = NP, then *MIN-FORMULA* ? P.

## Step-by-step solution

**Step 1** of 2

The two **Booleans** formulas are said to be **equivalent** if it consists the variables of the same set and the assignments of those variable on the same set become true. The definition of the minimal of the Boolean formula is given by "if no other sorter formula (Boolean) is equivalent to it".

• Consider a collection of minimal Boolean formulas, whose name is assigned as $MIN-FORMULA$. Therefore, the language defined for this $MIN-FORMULA$ can be given by:

$$MIN-FORMULA = \{\langle\varnothing\rangle \mid \varnothing \text{ is minimal, in the sense that if } \varnothing \equiv \varnothing', \text{ then } |\varnothing| \leq |\varnothing'|\}$$

Now, consider the language $NON-EQUIV$ which consists $\langle\varnothing,\varnothing'\rangle$ in such a way that $\varnothing$ and $\varnothing'$ are not equivalent.

• Now, it can be proved that $NON-EQUIV$ is in $NP$. To achieve this, consider a truth assignment is taken which is named as $c$. Now $V$ is taken in such a way that it is used to check $t(\varnothing) \neq t(\varnothing')$.

• So, $NON-EQUIV$ is in $P$ if $NP$ and $P$ are equivalent and also shows the closed property under complementation. Hence, it can be said that $NON-EQUIV$ is in $P$.

Comments (1)

**Step 2** of 2

Now, consider the $NOT-MIN-FORMULA$. Suppose a truth assignment is taken which is named as $c$, that defines a formula $\varnothing'$.

• Now $V$ is taken in such a way that it is used to check two things: $|\varnothing'| < |\varnothing|$ and $\varnothing' \equiv \varnothing$. Here, first condition gives the polynomial verifiability of second condition.

Now, from the above discussion **that describe the closeness property (**which say that, $P$ is closed under complementation**) of** $P$ **and equality of** $NP$ **and** $P$**, it can be said that the language** $MIN-FORMULA$ **is in** $P$**, if** $P = NP$**.**

Comment

## Problem

The ***difference hierarchy*** $D_iP$ is defined recursively as

**a.** $D_1P = NP$ and

**b.** $D_iP = \{A \mid A = B \setminus C$ for $B$ in NP and $C$ in $D_{i-1}P\}$.
(Here $B \setminus C = B \cap \overline{C}$.)

For example, a language in $D_2P$ is the difference of two NP languages. Sometimes $D_2P$ is called DP (and may be written $D^P$). Let

$$Z = \{\langle G_1, k_1, G_2, k_2\rangle \mid G_1 \text{ has a } k_1\text{-clique and } G_2 \text{ doesn't have a } k_2\text{-clique}\}.$$

Show that $Z$ is complete for DP. In other words, show that $Z$ is in DP and every language in DP is polynomial time reducible to $Z$.

## Step-by-step solution

**Step 1** of 3

Consider the difference hierarchy $D_iP$, which is defined recursively as

- $D_1P = NP\ and$
- $D_iP = \{A \mid A = B \cap \overline{C} \text{ for } B \text{ in } NP \text{ and } C \text{ in } D_{i-1}P\}$

Now consider the statement which is given below:

$Z = \{\langle G_1, k_1, G_2, k_2\rangle \mid G_1 \text{ has a } k_1 - \text{clique and } G_2 \text{ doesn't have a } k_2 - clique\}$

Comment

**Step 2** of 3

The above given statement ( $Z$ ) can be written in the form:

$Z = \{\langle G_1, k_1, G_2, k_2\rangle \mid \langle G_1, k_1\rangle \text{ in } CLIQUE \text{ and } \langle G_2, k_2\rangle in\ \overline{CLIQUE}\}$

• Suppose in DP, an arbitrary language is defined as $A = B \cap \overline{C}$. Any language is reducible in polynomial to $CLIQUE$ if they will be in $NP$.

• So, $B\ and\ C$ is polynomial reducible to $CLIQUE$. Hence, there exists a polynomial reduction function $S(B)$ and $S(C)$ which is used to reduce $B\ and\ C$ respectively.

• Both of the above functions output a coding like $\langle G, k\rangle$, where $k$ is defined as the clique size and $G$ is defined as a graph. So, the reduction ( $S(w)$ ) of both the function can be generated as $S(w) = S_B(w), S_c(w)$, which comprises a well definition of element of $Z \cup \overline{Z}$.

Comment

**Step 3** of 3

**Suppose** $w$ is in $B \cap \overline{C}$ then it shows that $S_B(w)$ is in $CLIQUE$ and $S_c(w)$ is in $\overline{CLIQUE}$. So that $S(w)$ will not be in $Z$. Hence, language $A$ will contain $w$ if and only if $S(w)$ in $Z$. As, $S(B)$ and $S(C)$ are polynomial and also $S(w)$ **shows polynomial behavior**. Therefore, $A$ **is polynomial reducible to** $Z$**. Hence it can be said that** $Z$ **is complete for DP.**

## Problem

Let *MAX-CLIQUE* = $\{\langle G, k\rangle \mid \text{ a largest clique in } G \text{ is of size exactly } k\}$. Use the result of Problem 7.47 to show that *MAX-CLIQUE* is DP-complete.

## Step-by-step solution

**Step 1** of 4

Consider the difference hierarchy $D_iP$, which is **defined** recursively as

- $D_1P = NP \text{ and}$
- $D_iP = \{A \mid A = B \cap \overline{C} \text{ for } B \text{ in } NP \text{ and } C \text{ in } D_{i-1}P\}$

Now consider the statement which is given below:

$Z = \{\langle G_1, k_1, G_2, k_2\rangle \mid G_1 \text{ has a } k_1 - \text{clique and } G_2 \text{ doesn't have a } k_2 - clique\}$

Comment

**Step 2** of 4

Here, it is already known that $Z$ **will be in** $DP$ **and every language in** $DP$ **is polynomial time reducible to** $Z$**.**

Comment

**Step 3** of 4

Now, consider the $NP$-completeness behavior of $MAX-CLIQUE$. To prove the given statement, first a $3-SAT$ problem will be reduced to $MAX-CLIQUE$.

• Particularly, a $m$ clause and $n$ variables, a $3-CNF$ formula $F$ will be generated. First for every clause $d$ of $F$, every assignment assigned to a variable c will be created as a node.

$F = (x_1 \vee x_2 \vee \overline{x}_4) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge \ldots$

Which show that there exist no edges between any two nodes of the same clauses.

• So it can be said that, the maximum clique size, that it shows, is $k$. It is well known that if a graph consist k-clique, then this graph will definitely acquire one node per clause $d$.

• Also, reduction which is taken will be in polynomial time. So, the produced graph shows the quadratic size of the graph.

• In other word it can be said that, it will take $F(O(k))$ nodes which consists $O(k^2)$ edges. **Therefore, it can be said that,** $MAX-CLIQUE$ **is** $NP$**-complete.**

Comment

**Step 4** of 4

**Therefore**, from the above discussion and from the definition of $DP$, **every language in** $DP$ **is polynomial time reducible to** $Z$ **and** $DP$ **is also in** $NP$**. Also,** $MAX-CLIQUE$ **is** $NP$**-complete. Hence it can be said that,** $MAX-CLIQUE$ **is** $DP$ **complete.**

Comment

## Problem

Let $f : \mathcal{N} \longrightarrow \mathcal{N}$ be any function where f(n) = o(n log n). Show that TIME(f(n)) contains only the regular languages.

## Step-by-step solution

**Step 1** of 4

$\text{Time} - \text{Compexity class } TIME(t(n)):$

Let $t : N \to R^+$ be a function. Define the time complexity class, $TIME(t(n))$, to be collection of all Languages that are decidable by an $O(t(n))$ time Turing Machine.

**Small - O – notation:**

Let $f$ and $g$ be functions $f, g : N \to R^+$. Say that $f(n) = O(g(n))$

If $\lim_{x \to \infty} \frac{f(n)}{g(n)} = 0$

In other words, $f(n) = 0(g(n))$ means that, form any real number $c > 0$, a number does not exists, where $f(n) < c(n)$ for all $n \geq n_0$.

Given that $f : N \to N$ be any function where $f(n) = 0(n \log n)$

• Now we have to show that $TIME(f(n))$ contains only regular languages.

• Suppose that $f(n) = 0(n \log n)$ and $M$ is a $S(\geq 2)$ state one – tape deterministic Turing machine accepting a set $L$ within time $f(n)$.

• Let $g(n)$ be defined by

$$g(n) = \begin{cases} \frac{n \log n}{f(n)}, & n \geq 2 \\ 1, & n = 0,1 \end{cases}$$

• Then we have $\lim_{x \to \infty} g(n) = \infty$ and we can select a value $c$ such that

$$3\frac{n^{(\log s)/g(n)^{\frac{1}{2}}+1} - 1}{S-1} + 1 \leq n - 2 - \frac{n}{g(n)^{\frac{1}{2}}} + C\frac{g(n)^{\frac{1}{2}}}{\log n}$$

For all $n \geq 2$.

• For this $c$, we show that the length of any crossing sequence of $M$ for any input $x$ in $L$ with $|x| \to 2$ is at most c.

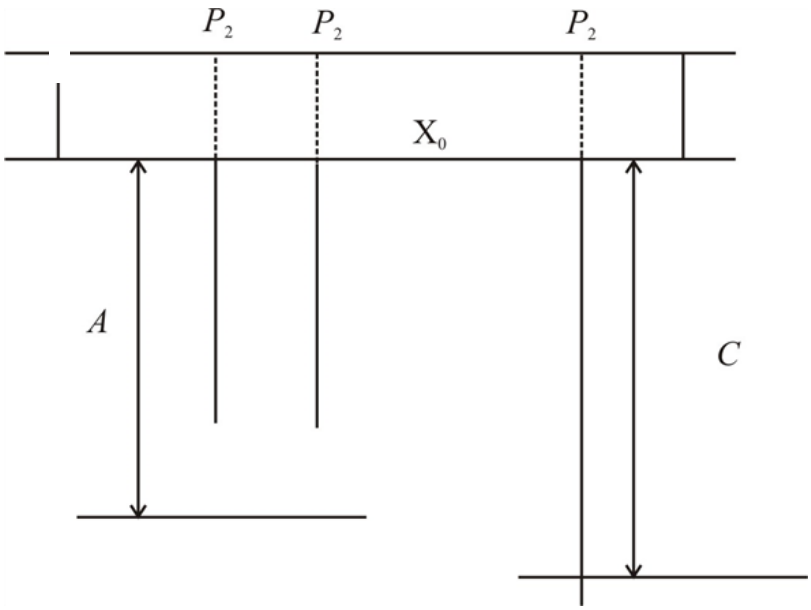
• Form this, it follows, that we can design a finite automaton that accepts $L$.

• Suppose that there is an $x$ in $L$ with $|x| \geq 2$ such that $M$ generates a crossing sequence of length larger than $c$ in accepting $x$.

• Let $x_0$ be the shortest such $x$, $n_0$ be its length, and $P_1$ be the position of one of such long crossing sequences.

Comment

**Step 2** of 4




**Figure**

Comment

**Step 3** of 4

In this figure, $A = \dfrac{\log n_0}{g(n_0)^{\frac{1}{2}}}$

• Suppose that $x_0$ was given to $M$.

• Let $h$ be the number of positions in $x_0$ (excluding both ends)

$(\log n_0)/\left(g(n_0)^{\frac{1}{2}}\right)$

• Then we have

$$\frac{n\log n_0}{g(n_0)} = f(n_0) > c + (n_0 - 2 - h)\frac{\log n_0}{g(n_0)^{\frac{1}{2}}},$$

And hence

$$h > n_0 - 2 - \frac{n_0}{g(n_0)^{\frac{1}{2}}} + c\frac{g(n_0)^{\frac{1}{2}}}{\log n_0}$$

$$\geq \quad 3\frac{n_0^{(\log s)/g(n_0)^{\frac{1}{2}}+1}}{S-1} + 1$$

$$= \quad 3\frac{S^{(\log n_0)/g(n_0)^{\frac{1}{2}}+1}}{S-1} + 1.$$

• Moreover, there are at most $= \dfrac{S^{(\log n_0)/g(n_0)^{\frac{1}{2}}+1} - 1}{S-1}$ crossing sequences of lengths smaller then $(\log n_0)/g(n_0)^{\frac{1}{2}}$

• Hence, at least four positions in $x_0$ have an identical crossing sequence.

• At least two of them are different from $P_1$ and are on the some side of $P_1$.

• Let $P_2, P_3$ be these positions (see figure 1).

• Let $x_0^1$ is the word obtained form $x_0$ by deleting the sub word between $P_2$ *and* $P_3$.

•

Comment

Then, $M$ accepts $x_0^{'}$, generating a crossing sequence of length larger than $c$ for $x_0^{'}$ and $2 \leq |x_0^{'}| < |x_0|$.

• This contradicts the selection of $x_0$.

Comment

## Problem

Call a regular expression ***star-free*** if it does not contain any star operations. Then, let $EQ_{SF-REX}$ = { $\langle R, S \rangle$ | R and S are equivalent star-free regular expressions}. Show that $EQ_{SF-REX}$ is in coNP. Why does your argument fail for general regular expressions?

## Step-by-step solution

**Step 1** of 2

Here in the given definition of language $\overline{EQ_{SF_RFX}}$ any string $s$ belongs to this iff it is of the following given form:

• Both of them from $R$ and $S$ are star free.

$s \in < R, S >,\ R$ and $S$ having both star free and $L(R) \neq L(S)$.

• $s$ do not represent any of two regular expression encoding.

• One of them from $R$ and $S$ is not star free or both are not star free.

$s \in < R, S >,$ either $R$ *or* $S$ are not star free or non of them are star free.

Comment

**Step 2** of 2

If $s$ do not represent any of two regular expression encoding or $s$ is of the form $< R, S >$ where either one is not star free or both are not star free then the string is easily accepted in DTM (Deterministic Turing Machine) in the polynomial time.

If first condition follows that is $R$ and $S$ are star free then there must be some string $u$ which is polynomial of size $|R| + |S|$. Hence there will be a NTM (Non Deterministic Turing Machine) which find string $u$ in polynomial time, but it will not able to find string $u$ where $L(R) = L(S)$.

Therefore, for general regular expression, argument fails. NTM recognizes $\overline{EQ_{SF_RFX}}$ in some polynomial time, Hence it prove the case.

Comment

## Problem

This problem investigates ***resolution***, a method for proving the unsatisfiability of cnf-formulas. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be a formula in cnf, where the $C_i$ are its clauses. Let $\mathcal{C} = \{C_i | \ C_i \text{ is a clause of } \phi\}$. In a *resolution step*, we take two clauses $C_a$ and $C_b$ in $\mathcal{C}$, which both have some variable $x$ occurring positively in one of the clauses and negatively in the other. Thus, $C_a = (x \vee y_1 \vee y_2 \vee \cdots \vee y_k)$ and $C_b = (\overline{x} \vee z_1 \vee z_2 \vee \cdots \vee z_l)$, where the $y_i$ and $z_i$ are literals. We form the new clause $(y_1 \vee y_2 \vee \cdots \vee y_k \ \vee \ z_1 \vee z_2 \vee \cdots \vee z_l)$ and remove repeated literals. Add this new clause to C. Repeat the resolution steps until no additional clauses can be obtained. If the empty clause ( ) is in C, then declare ? unsatisfiable.

Say that resolution is ***sound*** if it never declares satisfiable formulas to be unsatisfiable. Say that resolution is ***complete*** if all unsatisfiable formulas are declared to be unsatisfiable.

**a.** Show that resolution is sound and complete.

**b.** Use part (a) to show that *2SAT* ? P.

## Step-by-step solution

**Step 1** of 4

**a)**

The **Resolution** is defined as "it is proof procedure which uses **refutation**". In other words, "it can be defined as a formula which is used to proof that a formula is unsatisfiable by deriving $\perp$ from the formula". Resolution shows both of the behavior, that is, complete and sound.

- The **Resolution** is said to be **sound** if satisfiable formula can never be declared unsatisfiable by it.
- The **Resolution** is said to be **complete** if all the unsatisfiable formulas are declared to be unsatisfiable.

Comment

**Step 2** of 4

Now, consider **the lemma** which said that" if an application of resolution rule produces a clause $H$ (which is given as $H = H_1 \setminus \{L\} \cup C_2 \setminus \{\neg L\}$) under a valuation $Q$, then from the conjunction of the hypothesis of the rule, $H_1 \wedge H_2$, is false under $Q$".

- The above lemma can be proved by assuming $H$ is false under $Q$, but $H_1 \wedge H_2$ gives true value under $Q$. It is already given that $H_1$ shows a disjunction behavior, then one of its literals must be possess true under $Q$.
- All the literals other that $L$, then it is also exist in $H$ and therefore $H$ show true behavior, that shows a contradiction. Now, if $L$ is taken as a literal then $\neg L$ will be false under $Q$.
- It is already known that under $Q$, $H_2$ is true, then it must consists a literals other than $\neg L$ that shows true behavior under $Q$. But this show that it is also exists in $H$ and therefore $H$ shows a true nature under $Q$, that is a contradiction.

Thus, it can be said that $H_1 \wedge H_2$ must be false under $Q$. Now by using the concept of **induction**, it can be said that a **satisfiable formulas can never be declared as unsatisfiable.** Therefore, it can be said that **Resolution is sound.**

Comment

**Step 3** of 4

The **complete** property of **Resolution** can be proved by using the concept of induction. The induction will be applied on the excess number of literals.

- The excess number of literals in a clause is explained to be the number of literals, except in the clause. That is,

  $$excess(H) = \{0 \, if \, |H| < 2 \, and \, |H| - 1 \, if \, |H| \geq 2$$

- The number of excess literals in a clause set is just the sum of excess literals in every clause, that is,

  $$excess(H) = \sum_{i=1}^{n} excess(H_i)$$

Thus, an induction concept will be applied on the above. Therefore, it can be said that **Resolution is complete**.

Comment

**Step 4** of 4

**b)**

Consider **the decidability** of $2SAT$. It is known that $2SAT$ is **polynomial time decidable**. To proof this problem, efficiently a path searches in graphs can be used. A depth search or breadth search algorithm can be used in path search of graphs.

- If there exists an edge between $(a,b)$ then there must be exists a clause similar to $(\neg a, b)$ and also there exists a path between $(\neg b, \neg a)$.
- Now, consider a $2-CNF$ formula $\beta$. This formula is **unsatisfiable if and only if** there exists a variable $q$ in such a way that: "In graph, there exists a path from $q$ to $\neg q$ and also a path from $\neg q$ to $q$".
- Now consider there exists a path $q..\neg q$ and $\neg q..q$ fro a given variable $q$, but there exists also an assignment $\delta$ fro which $\delta(q) = T$ and similarly, $\delta(q) = F$.

Hence, from the above **explanation, it can be said that** $2SAT \in P$**.**

Comment

## Problem

Show that P is closed under homomorphism iff P = NP.

## Step-by-step solution

**Step 1** of 2

A **homomorphism** is defined as a function $f$ on strings with the property that $f(xy)=f(x)f(y)$. A **nonerasing homomorphism** is defined as a homomorphism $f$ such that $f(c)$ is not an empty string, for any character $c$. As it is known that both $\mathrm{NP}$ and $\mathrm{P}$ are closed under other operations, except for the homomorphism operation.

• To see both of the class $\mathrm{NP}$ **and** $\mathrm{P}$ **are not closed under homomorphism**, first of all initialize with a very hard language $\mathrm{L}$, which requires a time complexity of $2^{2^n}$.

• It can be made easy by appending each word of length exactly $2^{2^n}$ c's, where c denotes a new symbol. That is, suppose $\mathrm{L}'=\mathrm{C}^{2^2}$ then it can be said that $\mathrm{L}'$ is surely in $\mathrm{NP}$ and $\mathrm{P}$. But $\mathrm{L}=\mathrm{h}(\mathrm{L}')$, if $\mathrm{h}$ is defined as a homomorphism that send $c$ to $\varepsilon$ and is unique on all symbols of $\mathrm{L}$.

• If $\mathrm{NP}$ and $\mathrm{P}$ are closed under homomorphism, then $\mathrm{L}$ would be in $\mathrm{NP}$, which is not.

Comment

**Step 2** of 2

**Now**, the above explanation can be used to show $\mathrm{P}$ **is closed under homomorphism if and only if** $\mathrm{P}=\mathrm{NP}$**.** Suppose $\mathrm{h}^{-1}(\mathrm{L})$: every homomorphism $\mathrm{h}$ can only expand the length of the string to which it is applied by a constant factor.

• To recognize $\mathrm{h}^{-1}(\mathrm{L})$, apply $\mathrm{h}$ to the taken input $x$ and watch whether $\mathrm{h}(\mathrm{x})$ in $\mathrm{L}$. Now, see if there is a **polynomial time or nondeterministic polynomial-time test for membership in** $\mathrm{L}$**.**

• Now, the above test can be used in the same order for magnitude time complexity, which tells us whether, $x$ is in $\mathrm{h}^{-1}(\mathrm{L})$.

**Therefore**, from the above explanation, it can be said that " $\mathrm{P}$ **is closed under homomorphism if and only if** $\mathrm{P}=\mathrm{NP}$".

Comment

## Problem

Let $A \subseteq 1^*$ be any unary language. Show that if A is NP-complete, then P = NP. (Hint: Consider a polynomial time reduction f from *SAT* to A. For a formula ?, let $?_{0100}$ be the reduced formula where variables $x_1$, $x_2$, $x_3$, and $x_4$ in ? are set to the values 0, 1, 0, and 0, respectively. What happens when you apply f to all of these exponentially many reduced formulas?)

## Step-by-step solution

**Step 1** of 2

A language is defined as a **unary language** if it is subset of $1^*$ (at most $n$ strings of length $\leq n$). **A unary language is** $\mathrm{NP}$ **-complete then** $\mathrm{P=NP}$. The given statement can be shown by the following way:

Suppose $\mathrm{U} \subset 1^*$ be a unary language and let that $\mathrm{SAT} \leq \mathrm{U}$ through reduction $\mathrm{R}$. The function $\varphi$, which is defined as $\varphi(\mathrm{x}_1, x_2, \ldots, x_n)$ is an instance of $\mathrm{SAT}$.

Here, $\mathrm{SAT}$ **can be defined by the following algorithm:**

$\mathrm{SAT}(\varphi(\mathrm{A}))$

If $(|\varphi| == 1)$ return $\varphi$ *//trivial case True or False*

If $(A(f(\varphi)) ! = \text{undefined})$ return $A(f(\varphi))$

$A(f(\varphi)) = \mathrm{SAT}(\varphi(\mathrm{T}, x_2, \ldots, x_n)) \| \mathrm{SAT}(\varphi(\mathrm{F}, x_2, \ldots, x_n))$

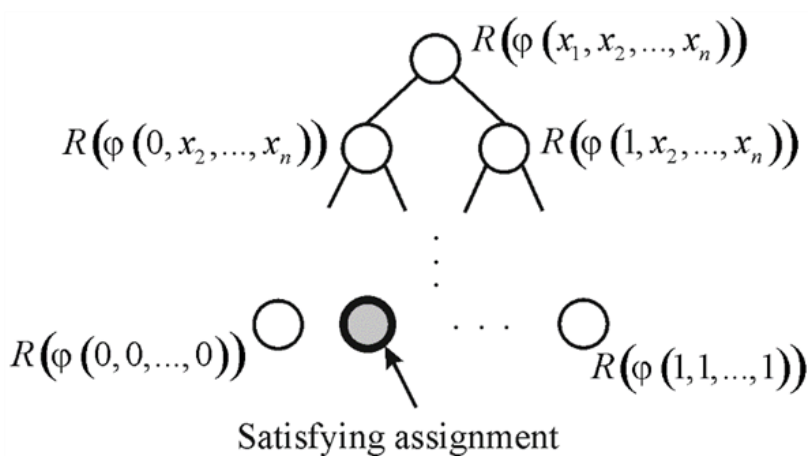
return $A(f(\varphi))$

The above algorithm explains how the self-reduction and reduction applied on the function $\varphi$ (which is further described below).

Comment

**Step 2** of 2

**Consider the figure which is given below:**




The above figure **explain self-reduction tree for** $\varphi$. Now, by the input of length $m = |\varphi(\mathrm{x}_1, x_2, \ldots, x_n)|$, $\mathrm{R}$ produces a string of length $\leq p(m)$.

• Here $\mathrm{R's}$ different outputs are colors. One color for the string which is not in $1^*$ and at most $p(m)$ for other colors .

• Hence, puzzle solution $\Rightarrow$ can solve $\mathrm{SAT}$ in $(\mathrm{p(m)+1, n+1}) = \mathrm{poly(m)}$ time!.

**Hence** from the above explanation it can be said that "**A unary language is** $\mathrm{NP}$ **-complete then** $\mathrm{P=NP}$."

Comment

## Problem

In a directed graph, the ***indegree*** of a node is the number of incoming edges and the ***outdegree*** is the number of outgoing edges. Show that the following problem is NP-complete. Given an undirected graph G and a designated subset C of G's nodes, is it possible to convert G to a directed graph by assigning directions to each of its edges so that every node in C has indegree 0 or outdegree 0, and every other node in G has indegree at least 1?

## Step-by-step solution

**Step 1** of 3

In a directed graph, the numbers of incoming edges are known as **in-degree** of a node and the numbers of edges, which are **out-degree,** are known as outgoing edges.

• Now, consider an undirected graph $G$ and a subset $C$ that consists of $G's$ node. The conversion of $G$ as a directed graph may be possible by assigning directions to each of its edges.

• This conversion is done in such a way that each node in $C$ consists an in-degree 0 or out-degree 0 and all other nodes in $G$ consists an in-degree of at-least 1. It can also be proved that the above problem is $NP-complete$.

Comment

**Step 2** of 3

First of all, consider $C=\varnothing$, then a graph is directed in such a way that all nodes consists a minimum degree of 1 ( $DA1$ ). It can be achieved by using the concept of directed Hamiltonian cycle ( $DHC$ ) and how the reduction takes place between $DHC$ to $DA1$.

• Suppose the undirected graph $G$ is given by $G=(V,E)$, where $V$ denotes the nodes and $E$ denotes the edges. Now, a graph $G'$ is created (which is directed) as $G'=(V,E')$, where $(u,v),(v,u)\in E'$ if $(u,v)\in E$.

• If the new graph consists a directed Hamiltonian cycle, that conclude that the real graph $G$ must consists a Hamiltonian cycle and vice versa.

Comment

**Step 3** of 3

Now, the directed Hamiltonian cycle can be reduced to the directed Hamiltonian path. Suppose a graph $G'=(V',E')$ is constructed from the given di-graph $G=(V,E)$. It is constructed by picking any vertex $v\in V$ and then break this in two vertices, which is given by $v_0$ and $v_1$.

• Suppose $V'=(V-v)\cup\{v_0,v_1\}$. Now, add an edge $(v_0,u)\in E'$, for every edge $(v,u)\in E$ and add an edge $(u,v_1)\in E'$, for every edge $(v,u)\in E$. Now, every other edges in $E$ is copied into $E'$.

• Hence, it is claimed that a Hamiltonian path exists that goes from $v_0$ to $v_1$ iff $G$ consists a directed Hamiltonian graph.

• Now, from the above discussion, $G'$ consists a Hamiltonian path from $v_0$ to $v_1$. Then, there exist an order of $E'$ that initialized from $v_0$ and terminated at $v_1$ and each is visited once.

• From the way of construction $E'$ takes place, it consists a path in $G$ which is initialized at $v$ and terminated at $v$ in such a way that it goes through each nodes $u\neq v$ perfectly once and $v$ exactly visited twice. Such a path in $G$ is known as its Hamiltonian cycle.

**Therefore, from the discussion it is concluded that "the problem given above is $NP-complete$ ".**

Comment

## Problem

Show that for any function $f : \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$, the space complexity class SPACE(f(n)) is the same whether you define the class by using the single tape TM model or the two-tape read-only input TM model.

## Step-by-step solution

**Step 1** of 3

$\mathrm{SPACE}(f(n))$: Let $f : N \to R^+$ be a function. The space completely class $\mathrm{SPACE}(f(n))$ is defined as follows:

$$\mathrm{SPACE}(f(n)) = \left\{ \begin{array}{l} L \mid L \text{ is a language decided by an } \mathrm{O}(f(n)) \text{ space} \\ \text{deterministic Turning machine} \end{array} \right\}$$

Now we have to prove that, class $\mathrm{SPACE}(f(n))$ is same whether we define $z$ by using the single tape *TM* or two tape read-only input *TM* model.

Comment

**Step 2** of 3

To prove this, we have to do the following two things

(i) We have to simulate single – tape *TM* on two tape read – only *TM* and

(ii) We have to simulate two tape read only *TM* on single – tape *TM*.

(i) **Simulation of single – tape *TM* on two tape read – only *TM:***

1. First we have to scan the read only tape.

2. Contents of the read only tape are copied to the work tape.

3. Remainder of the contents is simulated by treating the read l write work – tape as the input tape of a single – tape *TM*.

• Clearly only $\log n$ space is used to copy the contents.

• Since $f(n) \geq n$, we can write all of the input on are work – tape

In this way we simulated single-tape *TM* on two tape read only *TM*.

Comment

**Step 3** of 3

(ii) **Simulation two tape read only *TM* on single – tape *TM:***

• There is no participation from single tape TM; we are working over the first input symbols, using remainder of the tape as our work area.

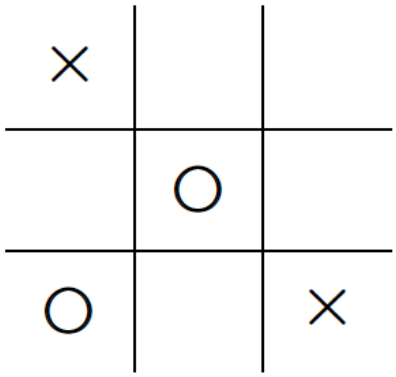
• For n symbols, add $n$ amount of space. By increasing space we add constant to $n$.

So from (i) and (ii) simulations we proved that class $SPACE(f(n))$ is same whether we defined it by using the single tape *TM* or two – tape read _ only input *TM*.

Comment

## Problem

Consider the following position in the standard tic-tac-toe game.



Let's say that it is the ×-player's turn to move next. Describe a winning strategy for this player. (Recall that a winning strategy isn't merely the best move to make in the current position. It also includes all the responses that this player must make in order to win, however the opponent moves.)

## Step-by-step solution

**Step 1** of 3

**Tic- tac – toe game:**

**Game Constraints:**

• This game is played by two players $X$ and $O$, who take turns marking the spaces in a $3\times3$ grid.

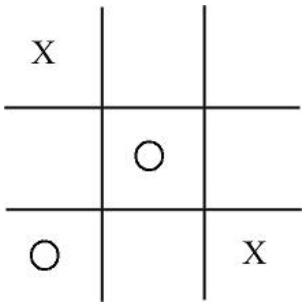
• Any $X$ or $O$ player goes first.

• **Winning the game:** who will first mark three places of vertical, horizontal or diagonal rows.

Comment

**Step 2** of 3

Consider $X$ moved first in following problem.




**Winning strategy for the game:**

The winning strategy of the game corresponds to quantified statement because games are closely related to quantifiers. These quantified statements help in understanding the complexity of the game.

Comment

**Step 3** of 3

Consider the quantified Boolean formula in prenex normal form, $\phi = \exists x_1 \forall x_2 \exists x_3 ... Q x_k [\psi]$. Here, *Q* is either $\forall$ or $\exists$. Two players called *X* and *O* selects the gr in the tic-tac-toe in their respective turn. Here, $x_1, x_2, x_3, \cdots$ are the grids in the tic-tac-toe. At this position of the ga e, three grids decide the winner.

The player *X* selects the grids that are bound to $\exists$ quantifier and player *O* selects the grids that are bound to $\forall$ quantifier. In this problem, player *X* starts the game. So, the formula starts with $\exists$ quantifier. The player *X* will win the game if $\psi$ is TRUE.

The formula is as follows:

$$\phi = \exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})]$$

In the above formula, $x_1$ denotes the grid selected by player *X*, $x_2$ denotes the grid selected by player *O* and $x_3$ denotes the grid selected by player *X*.

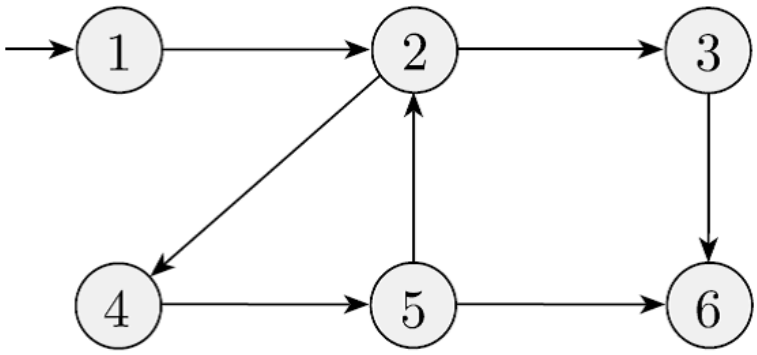
The player *X* always win the game if the third grid in the first row is selected (Consider it as 1 i.e., $x_1 = 1$)) and then selecting $x_3$ to be the negation of whatever player *O* selects.

The formula will become true, if the third grid in the first row is selected by player *X* then player *O* selects either third grid in the second row or the second grid in the first row or the first grid in the second row or the second grid in the third row. The player *X* can win the game if the second grid in the first row or the third grid in the second row is selected. Depending on the player *O*'s move, player *X* will select any of these grids and wins the game.

Comment

## Problem

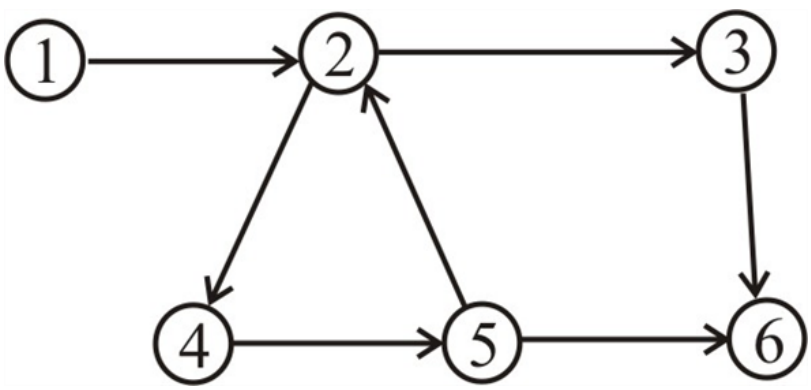
Consider the following generalized geography game wherein the start node is the one with the arrow pointing in from nowhere. Does Player I have a winning strategy? Does Player II? Give reasons for your answers.




## Step-by-step solution

**Step 1** of 3

Given generalized geography game is




Comment

**Step 2** of 3

Geography is a game contains nodes connecting. When one player begins at one node, another player continues with connecting the same nodes. If who will get stuck first i.e., there is no connection of node that player will lost the game.

For given problem, say that player I is the one who moves first and player II second.

Comment

**Step 3** of 3

**<u>Player I has a wining strategy as follows:</u>**

• Player I starts at node I, the designated start node.

• Node 1 points only at node 2, so player I must select node 2.

• Now Node 2 points at nodes 3 and 4.

• So player II must choose one of these two choices. He chooses node 3.

• Then player I must select 6, which doesn't point to any node.

• So player II is stuck, and thus player I win.

**Player II has a wining strategy as follows:**

• Player II starts at node 1, the designated start node.

• Node 1 points only at node 2, so player I must select node 2.

• Node 2 point at nodes 3 and 4.

• So player II must choose one of these two choices. He chooses node 4.

• Then player I must select node 5.

• Node 5 points at nodes 2 and 6.

• As node 2 already chosen, so player II must select node 6, which doesn't point to any node.

• So player I get stuck and thus player II wins.

Comment

## Problem

Show that PSPACE is closed under the operations union, complementation, and star.

## Step-by-step solution

**Step 1** of 4

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine i.e. $\mathrm{PSPACE} = \bigcup_k \mathrm{SPACE}\left(n^k\right)$.

Consider the two languages $L_1$ and $L_2$ that are decided by PSPACE Turing machines $M_1$ and $M_2$.

$M_1$ decides $L_1$ in deterministic time $O\left(n^k\right)$ and $M_2$ decides $L_2$ in deterministic time $O\left(n^L\right)$.

If any polynomial solvable in polynomial time, then it is solvable in polynomial space.

Comment

**Step 2** of 4

**UNION:**

$M =$"on input $w$:

1. Run $M_1$ on $w$, if $M_1$ accepted then accept

2. Else run $M_2$ on $w$, if $M_2$ accepted then accept

3. Else reject"

Clearly, the longest branch in any computation tree on input $w$ of length *n* is $O\left(n^{\max\{k,L\}}\right)$. Thus, $M$ is a polynomial time deterministic decider for $L_1 \cup L_2$. If any polynomial solvable in polynomial time, then it is solvable in polynomial space. Therefore, PSPACE is closed under union.

Comment

**Step 3** of 4

**COMPLEMENTATION:**

$M =$"on input $w$:

1. Run $M_1$ on $w$, if $M_1$ accepted then reject.

2. Else accept"

Clearly, the longest branch in any computation tree on input $w$ of length $n$ is $O\left(n^{\{k\}}\right)$. Thus, $M$ is a polynomial time deterministic decider for $\overline{L_1}$. If any polynomial solvable in polynomial time, then it is solvable in polynomial space. Therefore, PSPACE is closed under complementation.

Comment

**Step 4** of 4

**STAR:**

$M =$"on input $w$:

1. If $w = \varepsilon$ then accept

2. Deterministically select a number $m$ such that $1 \le m \le |w|$

3. Deterministically split $w$ into $m$ pieces such that $w = w_1 w_2 \ldots w_m$.

4. For all *i,* $1 \le i \le m$: run $M_1$ on $w_i$, if $M_1$ rejected then reject.

5. Else ( $M_1$ accepted all $w_i$, $1 \le i \le m$), accept".

The steps 1 and 2 takes $O(m)$ time. Step 3 also possible in polynomial time. In step 4, the for loop is run at most $m$ times and every run takes almost $O(m^k)$. The total times is $O(m^{k+1})$. This means that $M$ is a polynomial time decides for $L_1^*$. If any polynomial solvable in polynomial time, then it is solvable in polynomial space. Therefore, PSPACE is closed under star.

**Therefore, PSPACE is closed under Union, Complementation and star.**

Comment

## Problem

Show that $A_{DFA}$ ? L.

## Step-by-step solution

**Step 1** of 1

2287-8-7E AID: 1134 I 24/04/2012

RID: 48 I 18/05/2012

**Class *L* :** In deterministic Turing machine, L can be defined as class of languages complexity decidable in logarithmic space *i.e.* $L = SPACE(\log n)$

Now we need to prove that $A_{DFA} \in L$

$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a } DFA \text{ that accepts input string } w\}$

To prove $A_{DFA} \in L$ we need to construct a deterministic Turing machine (*TM*) to decide

$A_{DFA}$ in logarithmic space.

Let $M$ be the *TM* that decides $A_{DFA}$ in log – space.

The construction of *TM* $M$ is as follows:

$M =$"on input $\langle B, w\rangle$, where $B$ is a *DFA* and $w$ is string:

1. Simulate $B$ on $w$ by keeping track of $B$'s current state and its current head location and updating them appropriately.

2. If the simulation ends in an accept state, accept

3. else if the simulation end in non accepting that, reject"

The SPACE required to carry out this simulation is $O(\log n)$, since $n$ items of values storing its input by $M$. Thus we constructed *TM* $M$ to decide $A_{DFA}$ in log – space therefore $A_{DFA} \in L$.

Comment

## Problem

Show that any PSPACE-hard language is also NP-hard.

## Step-by-step solution

**Step 1** of 2

**PSPACE –completeness:** $A$ language $B$ is PSPACE – complete if it satisfies two conditions

1. $B$ exists in PSPACE, and

2. Every $A$ is PSPACE is polynomial time reducible to $B$.

If $B$ satisfies condition 2, we say that it is PSPACE – hard

Comment

**Step 2** of 2

**NP – Completeness** : A language $B$ is *NP* – complete if it satisfies two conditions.

1. $B$ is in *NP*, and

2. Every $A$ is *NP* is polynomial time reducible to $B$

If $B$ satisfies condition 2, we say that it is NP – hard.

Now we have to show that any PSPACE – hard language is NP – hard.

We know that NP is subset of PSPACE.

Therefore any string outside of PSPACE is also outside of NP.

In any problem from NP will reduced to PSPACE _ hard language.

• We know that "SAT = { <Φ>| is a satisfiable Boolean formula}."

• Also we know that "TQBF ={ < Φ>| is a true fully quantified Boolean formula}."

• We know that $SAT \in$ NP-complete

• Since any SAT problem can be reduced to a TQBF problem by simply adding "there exist $x_n$" to the front of SAT expression for each variable $x_n$.

So SAT problem can be solved using TQBF algorithm

• But *TQBF* problem is reduced to any *PSPACE* – hard problem. Because as we know that *TQBF* is *PSPACE* –complete.

• Thus *SAT* is reducing to *PSPACE* – hard, that means *NP* – hard problem is reduced to *PSPACE* – hard.

Thus any *PSPACE* – hard is also *NP* – hard.

Comment

## Problem

Show that NL is closed under the operations union, concatenation, and star.

## Step-by-step solution

**Step 1** of 4

**Class *NL(Non-deterministic Logarithmic space):***

*NL* is the class of languages that are decidable in logarithmic space on a non deterministic Turing machine. i.e., $NL = NSPACE(\log n)$

let $L_1$ and $L_2$ be the languages that are decided by *NL* - machines $M_1$ and $M_2$.

Now we want to show that

• There is a nondeterministic decides $M_\cup$ such that $L(M_\cup) = L_1 \cup L_2$

• There is a nondeterministic decider $M_\cap$ such that $L(M_\cap) = L_1 \cap L_2$ .

• There is a nondeterministic decider $M*$ such that $L(M_*) = L_1^*$

Now these 3 machines $M_\cup, M_\cap, M_*$ are for 3 different operations.

Comment

**Step 2** of 4

(i)

$M_\cup =$"on input $w$:

1. Run $M_1$ on $w$, If $M_1$ accepted then accept

2. Else run $M_2$ on $w$, if $M_2$ accepted then accept

3. Else reject"

The machine $M_\cup$ will accept the input $w$ if either $M_1$ or $M_2$ accept $w$.

If both $M_1$ and $M_2$ reject the input $w$ then $M_\cup$ will reject $w$.

Comment

**Step 3** of 4

(ii) **Intersection:**

$M_\cap =$"on input $w$:

1. Run $M_1$ on $w$, if $M_1$ rejected then reject.

2. Else run $M_2$ on $w$, if $M_2$ rejected then reject.

3. Else accept.

The machine $M_\cap$ will accept the input $w$ if both $M_1$ and $M_2$ reject $w$ then $M_\cap$ will reject $w$. If any one of $M_1$ and $M_2$ reject $w$ then $M_\cap$ will reject $w$.

Comment

**Step 4** of 4

(iii) **Star:**

Machine $M_*$ has more complex algorithm as follows.

$M_* =$ "on input $w$:

1. $P_1$ and $P_2$ be two input positions. $P_1$ is initialized to 0 and $P_2$ is initialized to the position immediately preceding the first input symbol.

2. If there is no input symbol after $P_2$ then accept $w$.

3. Move $P_2$ forward to a non-deterministically selected input position.

4. Simulate $M_1$ non-deterministically on the substring of $w$ from position following $P_1$ to the position $P_2$.

5. If $M_1$ is entered into accept state then copy $P_2$ to $P_1$ and go to stage 2

So $M_*$ will decide $L_1^*$,

In this way ***NL* is closed under union intersection and star**.

Comment

## Problem

Let $EQ_{REX}$ = { $\langle R, S \rangle$ I R and S are equivalent regular expressions}. Show that $EQ_{REX}$ ? PSPACE.

## Step-by-step solution

**Step 1** of 2

Language is $EQ_{REX} = \{(R,S) \mid R \text{ and S are equivalent regular expression}\}$

To show: $EQ_{REX} \in PSPACE$

PSPACE: PSPACE is deterministic Turing machine that contains the class of languages that are decidable in polynomial space on a deterministic Turing machine that is:

$$PSPACE = \bigcup_k SPACE(n^k)$$

For any language A, it is known that:

$$\begin{aligned} &\overline{A} \in NPSPACE \\ &\Rightarrow \overline{A} \in PSPACE \\ &\Rightarrow A \in PSPACE \end{aligned}$$

Thus, if it is shown that $\overline{EQ_{REX}} \in PSPACE$ then that implies that $EQ_{REX} \in PSPACE$

It is known that NPSPACE is non-deterministic Turing machine that contains the class of languages which are decidable in polynomial space.

Comment

**Step 2** of 2

Let M be the non-deterministic Turing machine that decides $\overline{EQ_{REX}}$ in a polynomial space as follows:

M= “On input (R, S) where R, S are equivalent regular expressions.” the following points are followed:

• Construct non-deterministic finite automata $N_x = (Q_x, \Sigma, \delta_x, q_x, A_x)$ such that $L(N_x) = L(X)$ for $X \in \{R,S\}$.

• Let $m_X = \{q_X\}$.

• Repeat $2^{\max} X \in \{R,S\}^{|Q_x|}$ times.

• If $m_k \cap A_s = \phi \Leftrightarrow m_s \cap A_s \neq \phi, accept$.

• Pick any $a \in \Sigma$ and change $m_x$ to $\bigcup_{q \in m_x} \delta_x(q,a)$ for $X \in \{R,S\}$.

• Reject

Hence, non-deterministic Turing machine M decides $\overline{EQ_{REX}}$ in polynomial space

Therefore, $\overline{EQ_{REX}} \in NPSPACE$ and hence $\overline{EQ_{REX}} \in PSPACE$

Comment

## Problem

A *ladder* is a sequence of strings $s_1, s_2, \ldots, s_k$, wherein every string differs from the preceding one by exactly one character. For example, the following is a ladder of English words, starting with "head" and ending with "free":

head, hear, near, fear, bear, beer, deer, deed, feed, feet, fret, free.

Let $LADDER_{\mathsf{DFA}} = \{\langle M, s, t\rangle \mid M$ is a DFA and $L(M)$ contains a ladder of strings, starting with $s$ and ending with $t\}$. Show that $LADDER_{\mathsf{DFA}}$ is in PSPACE.

## Step-by-step solution

**Step 1** of 3

$LADDER_{DFA} \in PSPACE$ $LADDER_{DFA}$= {(M, s, t) I M is a DFA and L(M) contains a ladder of strings, starting with s and ending with t}

The ladder is a sequence of strings where every string differs from preceding one in exactly one character.

**To prove:** $LADDER_{DFA}$ is in PSPACE

It is known that PSPACE=NPSPACE so, if it is shown that $LADDER_{DFA} \in NPSPACE$ then it implies that $LADDER_{DFA} \in PSPACE$

It is known that NPSPACE is non-deterministic Turing Machine that clears the class of languages which can be decided in polynomial space.

Comment

**Step 2** of 3

Follow the following steps:

• If $|s| \neq |t|$, then reject.

• Otherwise, the graph G of size vertices are indexed by strings in $2^{|s|}$ and there is a directed edge from $w_1$ to $w_2$ vary in precisely one character and $w_1, w_2 \in L(M)$

• $(M, s, t) \in LADDER_{DFA}$ if and only if there is a path from s to t in G.

• It can be checked in NPSPACE by guessing the path and at each step storing only the name of the current vertex.

• To guess the path at the vertex $w_1$ non deterministically select a new vertex $w_2$ that differs from $w_1$ in precisely one character and verify that M accepts $w_2$.

• The machine M always halts by keeping a counter and incrementing it with each guess and rejecting when the counter hits $|\Sigma|^{|s|}$

Comment

**Step 3** of 3

Thus, the non-deterministic Turing machine is constructed to decide $LADDER_{DFA}$

Therefore, $LADDER_{DFA} \in NPSPACE$ so, $LADDER_{DFA} \in PSPACE$

Comment

## Problem

The Japanese game *go-moku* is played by two players, “X” and “O,” on a 19 × 19 grid. Players take turns placing markers, and the first player to achieve five of her markers consecutively in a row, column, or diagonal is the winner. Consider this game generalized to an n × n board. Let

$$GM = \{\langle B\rangle \mid B \text{ is a position in generalized go-moku, where player “X” has a winning strategy}\}.$$

By a *position* we mean a board with markers placed on it, such as may occur in the middle of a play of the game, together with an indication of which player moves next. Show that *GM* ? PSPACE.

## Step-by-step solution

**Step 1** of 3

**PSPACE:** $\text{PSPACE}$ is deterministic Turing machine that contains the class of languages that are decidable in polynomial space on a deterministic Turing machine *i.e.,* $\text{PSPACE} = \bigcup_k SPACE(n^k)$

To generalize *go – moku* game on $n \times n$ board is given as

$GM = \{\langle B\rangle \mid B \text{ is a position in generalized } go\text{-}moku\text{, where player "X" has a winning strategy}\}$

• This game is play by 2 players $\text{"X" and "O"}$ Now we have to prove that $GM \in PSPACE$.

• Let us assume that $B$ is written as a grid of $\text{"X" and "O"}$ are empty, so the length of the input is $O(n^2)$

• Now let us define a recursive algorithm to solve $GM(B)$, which accepts if there is a winning strategy for player $X$ starting at position $B$.

Comment

**Step 2** of 3

**Recursive Algorithm of *GM* :**

$GM(B)$:

(1) $\text{potential } X \text{ moves:}$ All spaces of $i$ in position $B$ without marker on them

(i) Put an $X$ marker on space $i$, next changing the position to $B'$. If there are $5X's$ in a row (it a best move) then accept. If the board is now full, and no one has won, reject.

(ii) $\text{potential } O \text{ moves:}$ all spaces $j$ in position $B'$ without _ markers on them.

(a) Put an $O$ marker on space $j$, next changing the position to $B'$. if there are $5\,O's$ in a row(it is also best move) or the board in full and no one has won, loop to the next *i* (go to step(i)); putting an $X$ on $i$ is obviously a bad move.

(b) Otherwise, run $\text{GM}(\text{B}'')$

• If it accepts, loop to next $j$ (goto step (b)).

• If it rejects, loop to the next $i$ (goto step (i))

(iii) If all $j$ cause $\text{GM}(\text{B}'')$ to accept, $i$ is a good $X$ move, since it covers all possible $O$ moves, so accept.

Comment

**Step 3** of 3

(2) If no *i* step (i) causes accept, reject, there are no good moves from this position, so reject.

- We can loop through all moves $i, j$ at each step, since we can just reuse the space.
- As w just need to store configurations $B', B''$, which takes only $O(n^2)$ space.
- And our recursion is only $O(n^2)$ since there are at most $n^2$ moves.
- Total space needed is $O(n^4)$, which is a polynomial in the input length, since we assumed the input had length $O(n^2)$.
- Clearly no game of generalized *go – moku* on $n \times n$ board can have more than $n \times n$ moves.
- So possible configurations following from *B* needs only polynomial spac*e.*
- Thus $GM \in PSPACE$

Comment

## Problem

Show that if every NP-hard language is also PSPACE-hard, then PSPACE = NP.

## Step-by-step solution

**Step 1** of 1

Given statement,

Every *NP* – hard language is also $PSPACE-\text{hard}$. That means NP $\subseteq$PSPACE.

We have to show that

$PSPACE = NP$.

<u>*NP* – complete:</u> "A language *B* is *NP*- complete if it satisfies two conditions.

1. *B* is in *NP*, and

2. Every *A* in *NP* is polynomial time reducible to *B*"

If *B* satisfies condition 2, we say that it is *NP*- hard.

• From the hypothesis, Every *NP*- hard language is also *PSPACE*- hard.

• By the definition, *NP* – hard contains all of the *NP* – complete problem.

• So every *NP* – complete language is also *PSPACE*-hard. We know that *SAT* is *PSPACE*- hard.

• For any language *A* is *PSPACE,* A reduces to *SAT*.

• Assume that *A* is in *NP*.

Create a Turing Machine $(TM)$, *M* as follows

$M =$"On input *x*

1. compute $f(x)$. The poly – time nondeterministic algorithm between *A* and *SAT*.

2. If $f(x)$ is satisfible, accept *x*.

3. Else reject."

Claim *M* decides *A* since *x* is in *A* iff $f(x)$ is in *SAT*. Also *M* is an *NP* machine since computing *SAT* is in *NP*.

Thus, language *A* is in *PSPACE* then *A* is in *NP*

So, $PSPACE = NP$.

Comment

## Problem

Show that *TQBF* restricted to formulas where the part following the quantifiers is in conjunctive normal form is still PSPACE-complete.

## Step-by-step solution

**Step 1** of 3

*TQBF:* *TQBF* problem is to determine whether a fully quantified Boolean formula is true or false.

$TQBF = \{ \langle \phi \rangle \mid \phi$ is a true fully quantified Boolean formula$\}$

Show that *TQBF* restricted to formulas where the part following quantifiers is in conjunctive normal form $(cnf)$ is $PSPACE$- complete.

That is $cnf - TQBF = \{ \overline{Q}\phi \in TQBF \mid \phi \text{ is in } cnf \}$ is *PSPACE*- complete.

Comment

**Step 2** of 3

*PSPACE-* complete:

A language *B* is PSPACE- complete if it satisfies two conditions:

1. *B* is in *PSPACE,* and

2. Every *A* in *PSPACE* is polynomial time reducible to *B*.

If *B* merely satisfies condition 2, we say that it is *PSPACE*-hard.

1. $\underline{cnf - TQBF \in PSPACE}$: we know that $TQBF \in PSPACE$

As a subset of *TQBF* characterized by a simple syntactic test, $cnf - TQBF$ is obviously still in *PSPACE.*

2. $\underline{cnf - TQBF \in PSPACE - \text{hard}}$:

• We show $PSPACE - \text{hardness}$ by proving $TQBF \leq_p cnf - TQBF$

• Given a *TQBF* instance $\overline{Q}\phi$ where $\overline{Q}$ is a sequence of quantifiers and $\phi$ is a Boolean formula, we construct in polynomial time an equivalent $cnf - TQBF$ instance $\overline{Q}\overline{E}\psi$ where $\overline{E}$ is a sequence of existential quantifiers concerning the fresh proposition in $\psi$ but not in $\phi$.

• Here we use the technique for transforming the *SAT* instance $\phi$ in to an equi – satisfiable *CSAT* instance $\psi$.

• $\phi F \phi$ if and only if there exist an extension $\pi'$ of $\pi$ that make $\psi$ true.

• This construction establishes that

$\pi \models \phi$ that is $\pi \models \overline{E}\psi$ and hence $\models \overline{Q}\phi$ iff $\models \overline{Q}\,\overline{E}\psi$.

Comment

**Step 3** of 3

From (1) and (2) $cnf - TQBF$ is $PSPACE$ complete.

Thus, it is proved that the $TQBF$ restricted to formulas where the part following the quantifiers is conjunctive normal from, is still $PSPACE -$complete.

Comment

## Problem

Define $A_{LBA}$ = { $\langle M, w \rangle$ | M is an LBA that accepts input w}. Show that $A_{LBA}$ is PSPACE-complete.

## Step-by-step solution

**Step 1** of 4

$\underline{PSPACE-\text{complete}}$: A language *B* is *PSPACE* – complete if it satisfies two conditions.

1. *B* is in *PSPACE*, and

2. every *A* in *PAPACE* is polynomial time reducible to *B*.

If *B* satisfies condition 2, we say that *B* is *PSPACE*- hard

Comment

**Step 2** of 4

Given language is

$A_{LBA} = \{\langle M, w \rangle\} M \text{ is an } LBA \text{ that accepts input } w\}$

A linear bound automation $(LBA)$ is one – tape, one – head *NTM*.

We need to show that $A_{LBA}$ is $PSPACE-$complete.

That means $A_{LBA}$ has to satisfy the 2 conditions of *PSPACE*-complete.

Comment

**Step 3** of 4

(i) $\underline{A_{LBA} \in PSPACE}$:

To show $A_{LBA} \in PSPACE$, we need to construct a deterministic Turing machine that decides $A_{LBA}$ in polynomial space.

Let *T* be the Turing machine $(TM)$ that decides $A_{LBA}$ in polynomial space.

*T* can be constructed as follows.

$T =$" on input $\langle M, w \rangle$

Where *M* is a $TM(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ and $w \in \Sigma^*$

1. Take a step count *S* and initialize *S* to O.

2. While $S < |Q| \cdot |w| \cdot |\Gamma|^{|w|}$

(i) Simulate *M* on *w*

(ii) If *M* runs out of $|w|-$bounded tape then reject.

(iii) accept if *M* could in this step.

(iv) increment *S*.

3. Reject."

This machine *T* runs in polynomial space since the extra counter assumes values at most exponential in the length of the input word.

Thus we constructed a *TM T* to decide $A_{LBA}$ in polynomial space.

Therefore $A_{LBA} \subset PSPACE$

Comment

**Step 4** of 4

(ii) $\underline{A_{LBA} \text{ is } PSPACE-\text{hard}}$:

Let $L \in PSPACE$

Let $M$ be $TM$ that decides $L$ in space at most $n^k$

clearly $L \leq_P A_{LBA}$ by giving a reduction that maps $w$ to $\left\langle M, wL_1^{|w|^{k-1}} \right\rangle$.

Thus every $L$ in $PSPACE$ is polynomial time reducible to $A_{LBA}$.

Hence $A_{LBA}$ is $PSPACE$-hard.

From (i) and (ii) $A_{LBA}$ is $PSPACE-$complete

Comments (2)

## Problem

The cat-and-mouse game is played by two players, “Cat” and “Mouse,” on an arbitrary undirected graph. At a given point, each player occupies a node of the graph. The players take turns moving to a node adjacent to the one that they currently occupy. A special node of the graph is called “Hole.” Cat wins if the two players ever occupy the same node. Mouse wins if it reaches the Hole before the preceding happens. The game is a draw if a situation repeats (i.e., the two players simultaneously occupy positions that they simultaneously occupied previously, and it is the same player’s turn to move).

$$\begin{aligned} \textit{HAPPY-CAT} = \{\langle G, c, m, h\rangle \mid\ & G, c, m, h \text{ are respectively a graph, and} \\ & \text{positions of the Cat, Mouse, and Hole, such that} \\ & \text{Cat has a winning strategy if Cat moves first}\}. \end{aligned}$$

Show that *HAPPY-CAT* is in P. (Hint: The solution is not complicated and doesn’t depend on subtle details in the way the game is defined. Consider the entire game tree. It is exponentially big, but you can search it in polynomial time.)

## Step-by-step solution

**Step 1** of 1

Firstly note that game can have only $2n^2$ configuration, defined by position of the cat, position of the mouse and if it is cat's turn. So can construct a directed graph consisting of $2n^2$ nodes where every node corresponds to a configuration of game and there is an edge from node $u$ to the node $v$, if we can go from configuration corresponding to $u$ to configuration corresponding to $v$ in one move.

Now following algorithm solves HAPPY-CAT.

1. Mark all nodes $(a,a,x)$ where $a$ is a node in $G$, and $x \in \{true, false\}$.

2. If for a node $u=(a,b,true)$, there is a node $v=(c,b,false)$ which is marked and $(u,v)$ is an edge then mark $u$.

3. If for a node $u=(a,b,false)$, all nodes $v=(a,c,true)$ are marked and $(u,v)$ is an edge then mark $u$.

4. Repeat steps $2$ and $3$ until no new nodes are marked.

5. Accept if start node $s=(c,m,true)$ is marked.

Here algorithm takes $O(n^2)$ time to perform step 1, $O(n^2)$ time per iteration of the loop while loop is executed $O(n^2)$ times. Hence runs in polynomial time.

Comment

## Problem

Consider the following two-person version of the language *PUZZLE* that was described in Problem 7.28. Each player starts with an ordered stack of puzzle cards. The players take turns placing the cards in order in the box and may choose which side faces up. Player I wins if all hole positions are blocked in the final stack, and Player II wins if some hole position remains unblocked. Show that the problem of determining which player has a winning strategy for a given starting configuration of the cards is PSPACE-complete.

## Step-by-step solution

**Step 1** of 4

Consider the two-people version of the language $PUZZLE$. In this, every player uses an ordered puzzle card stack to start.

• The cards are placed in each turn in order, by the player, in the box and can select which part faces up. Player $I$ wins if every hole place are closed in the resultant stack and player $II$ wins if some position of the holes remains in unblocked state.

• Now, it can be shows that "the problem of determining which player has a winning strategy for a given starting configuration of the cards is $PSPACE$-complete".

Comment

**Step 2** of 4

Consider the formula $\varphi$, which is defined as:

$$\varphi = \exists p_1 \forall p_2 \exists p_3 \left[ (p_1 \vee p_2) \wedge (p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3}) \right]$$

In the above given formula for $\varphi$, player $I$ picks the value of $p_1$ and then player $II$ picks the value of $p_2$ and finally player $I$ picks the value of $p_3$.

• Now, assign true with the value 1 and false with the value 0. Suppose that player $I$ picks $p_1$=1, then the player $II$ picks $p_2$=0, and finally player $I$ picks $p_3$=1.

• Now by using these values in the sub-formula which is described above as

$$(p_1 \vee p_2) \wedge (p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3})$$

is1, therefore the player $I$ has won the game. Then, it can be said that "player $I$ has a strategy of winning for the given puzzle".

Comment

**Step 3** of 4

Now, a slightly change in formula, which is given above, will change the strategy of winning of the player.

• Consider the modified formula:

$$(p_1 \vee p_2) \wedge (p_2 \vee p_3) \wedge (p_2 \vee \overline{p_3})$$

The above formula is for winning strategy for the player $II$.

Comment

**Step 4** of 4

Consider the $PUZZLE$, which is described above, is $PSPACE-complete$. It is because, it is same as $TQBF$. The $PUZZLE$ problem is $PSPACE-complete$ can be proved by using the fact of $PUZZLE = TQBF$.

• Consider the formula $\varphi = \exists p_1 \forall p_2 \exists p_3 \ldots [\psi]$ became TRUE when setting of $p_1$ exists in such a way that, for every setting of $p_2$, a setting of $p_3$ exist is such a way and so on. Where, $\psi$ sows TRUE value under the setting of the taken variable.

• The same process will be applied for the winning strategy for the player $II$.

• If the formula $\varphi$ has of the form $\forall p_1, p_2, p_3 \exists p_4, p_5 \forall p_6 [\psi]$, player $I$ would make the first three move in e $PUZZLE$, to assign values to $p_1, p_2$ $and$ $p_3$.

• Then, player $II$ makes two moves to assign $p_4$ $and$ $p_5$. Finally, the player $I$ assigned a value $p_6$.

**Therefore, $\varphi \in TQBF$ exactly when $\varphi \in PUZZLE$. Hence it can be said that "the problem of determining which player has a winning strategy for a given starting configuration of the cards is $PSPACE$ -complete".**

Comment

## Problem

Read the definition of *MIN-FORMULA* in Problem 7.46.

**a.** Show that *MIN-FORMULA* $\in$ PSPACE.

**b.** Explain why this argument fails to show that *MIN-FORMULA* $\in$ coNP: If $\phi \notin$ *MIN-FORMULA*, then $\phi$ has a smaller equivalent formula. An NTM can verify that $\phi \in \overline{\textit{MIN-FORMULA}}$ by guessing that formula.

Problem 7.46

Say that two Boolean formulas are ***equivalent*** if they have the same set of variables and are true on the same set of assignments to those variables (i.e., they describe the same Boolean function). A Boolean formula is ***minimal*** if no shorter Boolean formula is equivalent to it. Let *MIN-FORMULA* be the collection of minimal Boolean formulas. Show that if P = NP, then *MIN-FORMULA* ? P.

## Step-by-step solution

**Step 1** of 1

a. Consider following algorithm

On Input $F$:

1. For each string $s$ such that $|s| < |F|$, if $s$ is a valid representation of a formula (this can be easily checked) which is equivalent to $F$ (this can be checked in polynomial space by evaluating both $F$ and $s$ over all possible truth assignments and comparing the results) then reject.

2. If all string has been tried without rejecting, accept.

Correctness of the algorithm should be evident. Only space used by the algorithm is for storing formula $F$, string $s$ and current assignment of literals which amounts to polynomial space. Hence, $MIN-FORMULA \in PSPACE$.

b. It fails to show $MIN-FORMULA \in coNP$ because it is not known if one can verify equivalence of two Boolean formulae is polynomial time.

Comment

## Problem

Let A be the language of properly nested parentheses. For example, (()) and (()(()))() are in A, but )( is not. Show that A is in L.

## Step-by-step solution

**Step 1** of 2

<u>The class L:</u> L is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is $L = SPACE(\log n)$

Given that

'*A*' be the language of property nested parenthesis.

For example, $((\ ))$ and $(((\ )))$ etc are in *A*. But not $)($.

Comment

**Step 2** of 2

We have to show that *A* is in *L*.

That means, we have to construct deterministic Turing machine $(DTM)$ that decides *A* in logarithmic space.

Let *M* be the *DTM* that decides *A* in logarithmic space.

The construction of *M* is as follows:

$M =$ "On input *w*:

Where *w* is a sequence of parenthesis.

1. Starting at the first character of *w*, move right across *w*.

2. when left parenthesis '(' is encountered, add 1 to the work – tape and move right.

3. when right parenthesis ')' is encounter and the work tape is blank, then reject.

Otherwise subtract 1 from the work – tape and move right.

4. When the end is reached, accept if the work tape is blank, reject if the work tape in not blank."

• Clearly, the only space used by this algorithm is for the counter on the work tape.

• If this counter is in binary, then the most space used by the algorithm is $O(\ln k)$

Where *k* is the number of $(s$.

• Since the number of $(s$ is less than or equal to *n* (the size of tape), this places the language *A* in *L*.

Thus we proved that $A \in L$

Comment

## Problem

Let B be the language of properly nested parentheses and brackets. For example, ([()()]()[]) is in B but ([)] is not. Show that B is in L.

## Step-by-step solution

**Step 1** of 3

The class $L$: $L$ is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is, $L = SPACE(\log n)$

Given that,

' $B$ ' be the language of property nested parenthesis. For example, $[(\ )(\ )]$.

But not $([)]$.

We have to show that $B$ is in $L$.

That means, we have to construct deterministic Turing machine $(DTM)$ that decides $B$ in logarithmic space.

Comment

**Step 2** of 3

Let *M* be the *DTM* that decides $B$ in logarithmic space.

The construction of *M* is as follows:

$M =$ "On input $w$:

Where *w* is a sequence of parenthesis and brackets.

1. if $w = \in$ then accept and halt.

2. Otherwise, take a counter $i = 0$.

3. Read the input sequentially from left to right.

4. When $[$ is read from the input, increment *i* by 1.

5. When $]$ is read from the input, decrement *i* by 1.

6. If *i* becomes negative or is not restored to zero at the end of the input, then reject.

7. Otherwise, for each symbol ' $a$ ' in the input sequentially from left to right repeat the following.

8. If a is $)$ or $]$, skip it.

9. Otherwise, ' $a$ ' is a left delimiter (or $[$, using a counter find our the matching right delimiter ' $b$ ' as we done from step 2 to step 6.

10. If $b$ is not found, or if $a$ and $b$ are of different types (parenthesis and bracket or bracket and parenthesis), then reject.

11. If not rejected so for then accept".

Comment

**Step 3** of 3

• Clearly the only space used by this algorithm is for the counter on the work tape.

• If this counters are binary, then the most space used by algorithm is $O(\ln k)$

• Where $k$ is number of brackets and parenthesis

• Since $k$ is always less than or equal to size of the tape this places the language $B$ in $L$.

• Thus we proved that $B \in L$.So, B is in $L$.

## Problem

The game of ***Nim*** is played with a collection of piles of sticks. In one move, a player may remove any nonzero number of sticks from a single pile. The players alternately take turns making moves. The player who removes the very last stick loses. Say that we have a game position in Nim with k piles containing $s_1, \ldots, s_k$ sticks. Call the position ***balanced*** if each column of bits contains an even number of 1s when each of the numbers $s_i$ is written in binary, and the binary numbers are written as rows of a matrix aligned at the low order bits. Prove the following two facts.

**a.** Starting in an unbalanced position, a single move exists that changes the position into a balanced one.

**b.** Starting in a balanced position, every single move changes the position into an unbalanced one.

Let *NIM* = { $\langle s_1, \ldots, s_k \rangle$ I each $s_i$ is a binary number and Player I has a winning strategy in the Nim game starting at this position}. Use the preceding facts about balanced positions to show that *NIM* ? L.

## Step-by-step solution

**Step 1** of 5

In a game of **NIM** that consists two players, there are $k$ heaps containing $s_1, s_2, \ldots, s_k$ numbers of stacks. Now, the binary equivalent of each of the numbers $(s_i)$ is taken and arrange them row wise and aligned them at the low order bits.

• The arrangement is said to be "balanced" if there are even no of one's in each column. An XOR operation is performed on the stick numbers in each heaps by the column values. **This is called "NIM-sum" of the numbers**.

• Hence, from the definition of balanced position, it can be said that if **zero value is acquired by the NIM-sum, then the arrangement is balanced otherwise not**.

Comment

**Step 2** of 5

Now to prove the given facts, consider that $s_1, s_2, \ldots, s_k$ be the heaps size before any move is made and $s_1', s_2', \ldots, s_k'$ be the size of heaps after a move is made.

• Assume $NIM_s$ be the NIM-sum before any move (that is, $NIM_s = s_1 \oplus s_2 \oplus \cdots \oplus s_k$) and $NIM_s'$ be the NIM-sum after any move (that is, $NIM_s' = s_1' \oplus s_2' \oplus \cdots \oplus s_k'$).

• If the move is in the heap $l$ then it has $s_i = s_i'$ for all $i \neq l$, and $s_l > s_l'$. It is considered that, the XOR function or more precisely in this case.

• The NIM-sum function $(\oplus)$ follow the simple associative and communicative laws and also follow one more property, that is, $s \oplus s = 0$.

Therefore, by using these properties, which are discussed above:

$$\begin{aligned} NIM_s' &= 0 \oplus NIM_s' \\ &= NIM_s \oplus NIM_s \oplus NIM_s' \\ &= NIM_s \oplus (s_1 \oplus s_2 \oplus \cdots \oplus s_k) \oplus (s_1' \oplus s_2' \oplus \cdots \oplus s_k') \end{aligned}$$

$$\begin{aligned} &= NIM_s \oplus (s_1 \oplus s_1') \oplus \cdots \oplus (s_k \oplus s_k') \\ &= NIM_s \oplus 0 \oplus 0 \oplus \cdots \oplus (s_l \oplus s_l') \oplus \cdots \oplus 0 \\ &= NIM_s \oplus s_l \oplus s_l' \end{aligned}$$

Comment

**Step 3** of 5

Consider an unbalanced position of piles (that is when $NIM_s \neq 0$), Suppose $d$ be the most significant position of non-zero bit in the binary format of $NIM_s$.

• Now, select a $l$ in such a way that a non-zero value is acquired by $d$th bit of $s_l$. The null or zero value acquired by the $d$th bit of $NIM_s$, when no such $l$ exists.

• Now consider, $s_l' = NIM_s \oplus s_l$ and $s_l' < s_l$ because all bits to the left of the $d$th bit is are same in $s_l$ and $s_l'$ and bit $d$ decreases from 1 to 0 or the value decreased by $2^d$.

• So the first player by the winning strategy makes a move by choosing $s_l - s_l'$ objects from heap $l$. Then

$$\begin{aligned} NIM_s' &= NIM_s \oplus s_l \oplus s_l' \\ &= NIM_s \oplus s_l \oplus (NIM_s \oplus s_l) \\ &= 0 \end{aligned}$$

**Therefore, there exists a single move that changes the position from unbalanced to a balanced one.**

Comment

**Step 4** of 5

Now, initialize from a balanced position, that is $NIM_s = 0$, then $NIM_s' = s_l \oplus s_l'$. Suppose, the first move is made on heap $l$ making $s_l \neq s_l'$ then $NIM_s'$ acquires a non-zero value**. Hence, every single move changes the position balanced to an unbalanced one.**

Comment

**Step 5** of 5

In a normal play condition, after considering the fact (that is, none of the players makes any mistakes) it is easy to find a winning strategy. It is clear from the above mathematical deduction, that the strategy is to make the NIM-sum zero after each move.

• Consider the winning strategy, if the game is in balanced condition (that is NIM-sum is already zero). User has to move second letting Player II to move first and conversely when the game condition is unbalanced one (that is, the NIM-sum is non-zero). Then player I's has to move first.

• Now considering an algorithm to design a TM for Player I's winning strategy, where each input is already in its binary equivalent. These easy steps can be followed:

Suppose, M = Step

Cross off the 1's on finding of two 1s.

a. If no 1 left uncrossed after the entire scan of input, Player-I has to move second.

b. Else, Player-I has to move first.

**So, the only space tradeoff is to store the binary numbers, which requires $O(\log(k))$ space. Therefore,** $NIM \in L$. □

Comment

## Problem

Let *MULT* = {a#b#cl a, b, c are binary natural numbers and a × b = c}. Show that *MULT* ? L.

## Step-by-step solution

**Step 1** of 2

The class $L$: L is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is, $L = SPACE(\log n)$

Given that,

$MULT =\{$ $a\#b\#c \mid$ where $a,b,c$ are binary natural numbers and $a \times b = c\}$.

Comment

**Step 2** of 2

We have to show that $MULT \in L$.

That means, we have to construct a deterministic Turing machine $(DTM)$ that decides $MULT$ in logarithmic space.

Let $M$ be the *DTM* that decides *MULT* in logarithmic space.

The construction of $M$ is as follows:

$M =$"On input $a\#b\#c$:

1. If either of the three strings is not a binary number as defined above then reject

2. Initialize a binary counter *i* pointing to the O.

3. Initialize *a* binary counter *j* to $\max(0, i+1-\text{length of } 1)$

4. Initialize a binary counter *k* to $i-j$

5. Now take a binary counter $x \leftarrow x + a[j]*b[k]$

6. Repeat steps 3 to 5 $\min(i, n-1)$ times.

7. Repeat steps 2 to 6 $2n-1$ times and calculate $x = \text{floor}(x/2)$

8. If any discrepancy arises between the calculate next bit of *c* and then reject.

9. If the multiplication ended with no errors then accept.

Thus, clearly *M* runs in log space and decides $MULT$.

So, $MULT \in L$.

Comment

## Problem

For any positive integer x, let $x^R$ be the integer whose binary representation is the reverse of the binary representation of x. (Assume no leading 0s in the binary representation of x.) Define the function $\mathcal{R}^+ : \mathcal{N} \longrightarrow \mathcal{N}$ where $\mathcal{R}^+(x) = x + x^{\mathcal{R}}$.

**a.** Let $A_2 = \{\langle x, y\rangle \mid \mathcal{R}^+(x) = y\}$. Show $A_2 \in \mathrm{L}$.

**b.** Let $A_3 = \{\langle x, y\rangle \mid \mathcal{R}^+(\mathcal{R}^+(x)) = y\}$. Show $A_3 \in \mathrm{L}$.

## Step-by-step solution

**Step 1** of 1

**Given:**

$x$ is one of the positive integer and the reverse of the integer $x$ is donated by the symbol $x^R$ in the binary representation. But the condition is that there is no zero's in the binary representation of the number *x*.

Function $R^+$ need to be defined in such a way that the integer $x$ is a natural number.

$R^+(x) = x + x^R$

Suppose, the positive integer $x$ is 15, binary representation of $(15)_{10}$ is written as $(1111)_2$.

The reverse of binary representation $x^R$ is represented as $(0000)_2$.

Inverse of the binary representation is done by converting the ones with zeros and zeros with ones.

**Proof:**

Consider the Turing machine *M* for every positive integer *x* , inverse of *x* is computed. When the computation is performed on the integer *x* and the inverse of the integer *x* then result will also be *x.*

This is because when a binary number 1 is added to 0 then result is always 1. Here, *x* positive integer is converted into binary representation but the condition is that the binary representation of *x* should not contain 0.

Turing machine will accept only those values of *x* whose binary representation is 1. Even Turing machine will accept those values of inverse of *x* whose binary representation is 0.

After that machine perform the computation between *x* and $x^R$.

Hence, it is proved that language $A_2 \in L$.

Consider the Turing machine *M* for every positive integer *x* , inverse of *x* is computed. When the computation is performed on the integer *x* and the inverse of the integer *x* then result will also be x.

This is because when a binary number 1 is added to 0 then result is always 1. Here, *x* positive integer is converted into binary representation but the condition is that the binary representation of *x* should not contain 0.

After performing the computation Turing machine once again call the same function. After calling the same function also the result after computation will be the value equal to *x.*

Turing machine will accept only those values of *x* whose binary representation is 1. Even Turing machine will accept those values of inverse of *x* whose binary representation is 0.

After that machine perform the computation between *x* and $x^R$.

Hence, it is proved that language $A_3 \in L$.

Comment

## Problem

**a.** Let *ADD* = { $\langle x, y, z \rangle$ | x, y, z > 0 are binary integers and x+ y = z}. Show that *ADD* ? L.

**b.** Let *PAL-ADD* = { $\langle x, y \rangle$ | x, y > 0 are binary integers where x + y is an integer whose binary representation is a palindrome}. (Note that the binary representation of the sum is assumed not to have leading zeros. A palindrome is a string that equals its reverse.) Show that *PAL-ADD* ? L.

## Step-by-step solution

**Step 1** of 2

The class $L$: $L$ is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is, $L = SPACE(\log n)$

(a) Given Language is

$ADD = \{\langle x, y, z \rangle \mid x, y, z > 0 \text{ are binary integers and } x + y = z\}$

We have to show that $ADD \in L$.

That means, we have to construct a deterministic Turing machine $(DTM)$ that decides *ADD* in logarithmic space.

Let $M_1$ be the *DTM* that decides *ADD* in logarithmic space.

The construction of $M_1$ is as follows:

$M_1 =$"On input $\langle x, y, z \rangle$:

1. If either of the three strings is not a binary number in the sense defined above then reject.

2. Initialize three binary counters $i, j, k$ pointing to the right – most of $x, y$ and respectively.

3. Perform binary addition (bit – wise long addition) using $i, j, k$ and *a* carry flag.

4. If any discrepancy arises between the calculated next bit of *z* and the actual next bit of *z*, then reject.

5. If the end of all three number representations is reached and no error detected, then accept. So clearly $M_1$ runs in log space (it uses 3 counter only) and decides *ADD*".

Thus *ADD* is in *L*. So, $ADD \in L$.

Comment

**Step 2** of 2

(b) Given language is

$PAL_ADD =$ { $\langle x, y \rangle \mid x, y > 0$ are binary integers where $x + y$ is an integer whose binary representation is a palindrome}

We have to show that $PAL_ADD \in L$.

That means, we have to construct a *DTM* that decides $PAL_ADD$ in logarithmic space.

Let $M_2$ be the *DTM* that decides $PAL_ADD$ in logarithmic space.

The construction of $M_2$ is as follows.

$M_2 =$"On input $\langle x, y \rangle$:

1. If either of the two strings is not a binary number in the sence defined above, then reject.

2. Compute the length *l* of $x + y$ in binary.

3. For $i = 1$ to $l/2$.

(i) Compute the i$^{th}$ bit a of $x + y$

(ii) Compute the $(l-i+1)^{th}$ bit *b* of $x+y$

(iii) if $\neq b$ then reject.

4. otherwise, accept"

Clearly $M_2$ runs in logspace and decides $PAL_ADD$.

Thus $PAL_ADD$ is in L. So, $PAL_ADD \in L$

Comment

## Problem

Define *UCYCLE* = { $\langle G \rangle$ | G is an undirected graph that contains a simple cycle}. Show that *UCYCLE* ? L. (Note: G may be a graph that is not connected.)

## Step-by-step solution

**Step 1** of 3

The class $L$: L is the class of languages that are decidable in logarithmic space on a deterministic truing machine.

That is, $L = SPACE(\log n)$

Given language is

$UCYCLE =$ { $\langle G \rangle \mid G$ is an undirected graph that contains a simple cycle}

We have to show that $UCYCLE \in L$.

Comment

**Step 2** of 3

Let *M* be the deterministic Turing machine that decides $UCYCLE$ the construction of $M$ is as follows:

$M =$ "On input $\langle G \rangle$ ( *G* is an undirected graph):

1. Select a vertex *u* as starting vertex

2. Select an edge $(u,v)$ from *u*.

3. Start traversal through $(u,v)$, if we come back to *u* through an edge different that $(u,v)$, then accept.

4. Otherwise, reject"

Comment

**Step 3** of 3

• If we come back to the starting vertex through an edge different than the one we started on, we declare that the graph contain a cycle.

• Since all the vertices and all the edges are enumerated in logspace *M* decides $UCYCLE$ in logarithmic space.

• Therefore, $UCYCLE \in L$.

Comment

## Problem

For each n, exhibit two regular expressions, R and S, of length poly(n), where L(R) ≠ L(S), but where the first string on which they differ is exponentially long. In other words, L(R) and L(S) must be different, yet agree on all strings of length up to $2^{\epsilon n}$ > 0.

## Step-by-step solution

**Step 1** of 2

Consider that two regular expressions, $S$ and $R$, of length $poly(n)$ can be exhibited for every $n$,where $L(R) \neq L(S)$but the first string on which they differ in exponentially long.

• Now, consider the expression the expression which is given below:

$$\underbrace{(11...1)}_{p\,\text{times}}^{*}\underbrace{11?1?...1?}_{p-2\,\text{times}}$$

**The above expression identify $1^k$ for each $k$ which is not a multiple of $p$ .**

Comment

**Step 2** of 2

Now, **a polynomial-length** can be constructed in such a way that recognize $1^k$ for each $k$ and except those expression which are multiple of every first $n$ prime.

• The $n$th prime is less than $n(\ln n + \ln\ln n)$foe every $n$ ≥6. Therefore, the addition of the first $n$primes is $O(n^2 \log n)$.

• It is because the first number which is not a multiple of any of the first prime number is of the order $O(n^2 \log n)$, that cause to stop it in growing exponentially with $n$.

Hence form the above explanation, it can be said that "**every string of the length up to $2^{\varepsilon n}$, for a constant $\varepsilon > 0$, will be agreed for two regular expression $S$ and $R$, where $L(R)$and $L(S)$must be different".**

Comment

## Problem

An undirected graph is ***bipartite*** if its nodes may be divided into two sets so that all edges go from a node in one set to a node in the other set. Show that a graph is bipartite if and only if it doesn't contain a cycle that has an odd number of nodes. Let *BIPARTITE* = { $\langle G \rangle$ I G is a bipartite graph}. Show that *BIPARTITE* ? NL.

## Step-by-step solution

**Step 1** of 4

Given that

$BIPARTITE = \{ \langle G \rangle \mid G$ is a bipartite graph}

An undirected graph is bipartite if its nodes may be divided into two sets so that all edges go from a node in one set to a node in the other set.

Now we have to show that

• A graph is bipartite if and only if it doesn't contain a cycle that has an odd number of nodes, and

• $BIPARTITE \in NL$

Comment

**Step 2** of 4

(i) <u>First we show that if a graph is bipartite then it doesn't contain a cycle that has an odd number of nodes:</u>

• Let *G* be a graph which is bipartite.

• Assume that *G* contain a cycle that has odd number of nodes

• $n_1, n_2, n_3 \ldots, n_{2k}, n_{2k+1}$

• By the definition of bipartite graph, as *G* is a bipartite we have $n_1$ is in some set *A*, and $n_2$ must be in some set *B*, then $n_3$ must be in set *A* etc.

• By induction, all the nodes with an odd subscript must be in set *A*, and all those with an even subscript must be in set *B*

• This implies that $n_1$ and $n_{2k+1}$ are both in set *A*.

• This is a contradiction because they are connected.

• Thus our assumption that *G* contains a cycle that has odd number of nodes is wrong.

• Hence *G* does not contain a cycle with odd number of nodes.

Comment

**Step 3** of 4

(ii) <u>Second, if a graph doesn't contain a cycle with an odd number of nodes then the graph is bipartite:</u>

• Suppose a graph does not contain any odd cycles.

• Pick a node, and label it *A*.

• Label all of its neighbors *B*.

• Label all of their unlabeled neighbors *A*, etc, until all nodes are labeled.

• Suppose that this construction caused two adjacent nodes *x* and *y* to have the same label.

• Then that would mean that both *x* and *y* were reached by taking an even number of steps.

• In either case, the total number of nodes traversed in getting to *x* from the start node, and getting to *y* from the start node (excluding the start node itself), is even.

• But adding the start node in makes the total number of nodes odd, contradicting the hypothesis that there were no odd cycles.

• Thus, the construction succeeds in properly dividing the nodes, so the graph is bipartite.

Comment

**Step 4** of 4

(iii) Finally, we show that $BIPARTITE \in NL$

• We know that $NL = CONL$

• So if we prove that $\overline{BIPARTITE} \in NL$ then than implies that $BIPARTITE \in NL$

• $\overline{BIPARTITE}$ ={ $\langle G \rangle \mid G$ is a graph that contain an add cycle}

We know that

" *NL* is the class of languages that are decidable in logarithmic space on a non deterministic Turing machine "

Let *M* be the *NTM* that decides $\langle G \rangle \mid G$ $\overline{BIPARTITE}$ in logarithmic space. *M* can be constructed as follows:

$M =$ "on input $\langle G \rangle$:

1. Counter : = 0

2. start: = Nondeterministically select a starting node.

3. Successor := Nondeterministically select a successor of *s*.

4. while (counter $\leq$ the number of nodes)

5. If successor = start and if counter is odd accept

6. otherwise successor $:=$ nondeterministically select a successor of successor

7. If counter has increased above the number of nodes, reject."

Since, the only space used by the above algorithm is for keeping track of the node numbers 'start' and 'Successor' and keeping track of the 'counter', the algorithm clearly only uses log space on any one branch.

• There fore $\overline{BIPARTITE} \in NL$

• Because $NL = CONL, BIPARTITE \in NL$

Comment

## Problem

Define *UPATH* to be the counterpart of *PATH* for undirected graphs. Show that

$\overline{BIPARTITE} \leq_L UPATH$. (Note: In fact, we can prove $UPATH \in L$, and therefore $BIPARTITE \in L$, but the algorithm [62] is too difficult to present here.)

## Step-by-step solution

**Step 1** of 2

An assumption $UPATH \in L$ can be used to show $BIPART \in L$ . **Now,** As it is known that, all paths which contain a length of two in $G$ are used to acquire the graph of $G^2$. To show the bipartite condition, all graphs have to show a certain criteria. One of the criteria is that all the graphs should consist a cycle of odd length and also an edge between $p$ and $q$ or an edge $(p,q)$. By considering the above explanation, $G$ can be said as bipartite iff $\forall$ edges $(p,q)$ of $G$, there is no connection between $p$ and $q$ by the path taken in $G^2$. Consider the algorithm which is given below shows how a logarithmic space can be used to solve $BIPART$.

*//algorithm*

**On input** $G$

**For every edges** $(p,q)$ **in** $G$

*// check for the existence of path.*

**"Check if there is a path exists from** $p$ **to** $q$ **in** $G^2$**"**.

Now, the logarithmic space can be used to implement whole the procedure discussed above and **this logarithmic space can be achieved by checking if there are edges between** $(u,r)$ and $(r,q)$.

Comment

**Step 2** of 2

Now, $UPATH \in L$ **can be shown by taking an assumption** $BIPART \in L$. The condition of bipartite can also be checked by the color of the vertices (which is generated by any algorithm). Now, consider the graph algorithm (which is used to generate the graph) which is given below:

For all vertex $q$ of $G$

"**Two copy of** $q_1, q_2$ **is made in** $H$**"**.

For all edge $(q,r)$ of $G$

"**Put the edges** $(q_1,r_2)$ **and** $(q_2,r_1)$ **in** $H$".

By using the facts which is discussed above, $H$ will be **bipartite** only if every vertices of $q_1$ and $q_2$ are colored red and blue respectively. It gives that there exists no monochromatic edge. So, a cycle of odd length cannot exist. Suppose, a moment $(G,m,n)$ of $BIPART$ is given.

• Now, by connecting the edges $(m_1,n_1)$ and $(m_2,n_2)$, the graph $H_1$ and $H_2$ can be obtained respectively. If there exists a path from $m$ to $n$ in $G$, then minimum one of the graphs $H_1$ and $H_2$ **must consists a cycle of odd length**.

• However, if $n$ and $m$ are not joined in $G$, then both $H_1$ and $H_2$ will show bipartite behavior. In $H_1$, $m_1$ and $n_1$ will be in different component.

• Since, bipartite behavior is shown by both of these components, it exists also after adding the edge $(m_1,n_1)$ and acquiring $H_1$, the graph remains bipartite**. In the same way,** $H_2$ **must be bipartite.**

**Therefore, it can be said that** $UPATH \in L$ **and therefore,** $BIPART \in L$ **or In other words "** $\overline{BIPARTITE} \leq_L UPATH$**"**.

Comment

## Problem

Recall that a directed graph is ***strongly connected*** if every two nodes are connected by a directed path in each direction. Let

$$STRONGLY\text{-}CONNECTED = \{\langle G\rangle \mid G \text{ is a strongly connected graph}\}.$$

Show that *STRONGLY-CONNECTED* is NL-complete.

## Step-by-step solution

**Step 1** of 4

Specified that

A directed graph is strongly connected if every two nodes are connected by a directed path in each direction.

Let $STRONGLY_CONNETED = \{\langle G\rangle \mid G$ is a strongly connected graph}

We have to show that $STRONGLY_CONNETED$ is $NL_$complete

<u>*NL* -completeness:</u> A language '*B*' is *NL*_ complete if

1. $B \in NL$, and

2. Every *A* in *NL* is log space reducible to *B*.

So, to show that $STRONGLY_CONNECTED$ is *NL* _ complete

We need to prove the 2 conditions of $NL_$completeness.

Comment

**Step 2** of 4

(1) $\underline{STRONGLY_CONNECTED \in NL}$:

We know that

"*NL* is the class of languages that are decidable in logarithmic space on non-deterministic Turing machine $(NTM)$,"

So to prove $STRONGLY_CONNECTED \in NL$, we need to construct a *NTM N* that decides $\overline{STRONGLY_CONNECTED}$ in logarithmic space.

The construction of *N* is as follows:

$N_1 =$ "On input $\langle G\rangle$:

1. Select two nodes *a* and *b* non-deterministically.

2. Run $PATH(a,b)$.

• If it rejects, then the graph is not strongly connected, so accept.

• Otherwise, reject.

Since storing the node numbers *a* and *b* only takes log space, and *PATH* uses only log space, so

$\overline{STRONGLY_CONNECTED} \in NL$.

We know that $NL = CONL$.

Thereof $STRONGLY_CONNECTED \in NL$

Comment

**Step 3** of 4

(2) Next we must show that every language in *L* is log space reducible to $STRONGLY_CONNECTED$.

We do this by reducing $PATH$ to $STRONGLY_CONNECTED$

The $NTM$ $N_2$ will do this procedure.

$N_2 =$ "On input $\langle G,s,t \rangle$, where *G* is a graph and *s, t* are vertices in *G*

1. Copy all of *G* onto the output tape
2. Additionally for each node *i* in *G*.
3. Output on edge from *i* to *s*.
4. Output an edge from *t* to *i.* "

Comment

**Step 4** of 4

This algorithm only needs log space to store the counter for *i*

• If there is a path from *s* to *t*, then the constructed graph is strongly connected because every node can now get to every other node by going through the path $s-t$.

• If there is no path from *s* to *t*, then the graph is not strongly connected. Because the only additional edges in the constructed graph go into *s* and out of *t* , so there can be no new ways of getting from *s* to *t*.

So, from (1) and (2)

$STRONGLY_CONNECTED$ is *NL* complete

Comment

## Problem

Let

$$BOTH_{\text{NFA}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are NFAs where } L(M_1) \cap L(M_2) \neq \emptyset\}.$$

Show that $BOTH_{\text{NFA}}$ is NL-complete.

## Step-by-step solution

**Step 1** of 3

Consider the given language, $BOTH_{NFA} = \{\langle M_1, M_2\rangle | M_1 \text{ and } M_2 \text{ are NFAs where, } L(M_1) \cap L(M_2) \neq \varnothing\}$. A non-deterministic log space algorithm is given here, which determines that "there are some strings which both $M_1$ and $M_2$ accepts". Consider that $q_1$ and $q_2$ are the number of states in both NFAs accordingly.

$M =$ "On input $\langle M_1, M_2\rangle$, where $M_1, M_2$ are NFAs:

1. Place markers on the start states of both of the NFAs.

2. Repeat $2^{q_1 \times q_2}$ times:

3. In each step select non-deterministically an input symbol and accordingly change the marker positions on $M_1$ and $M_2$'s states to simulate reading of that symbol.

4. **Accept;** if some string is found in stage 2 and 3 on which both $M_1$ and $M_2$ attends any accepting state that is if at some point on input string both the markers are on accepting states of $M_1$ and $M_2$. Otherwise **Reject**."

The only space requirement for this algorithm is to store the location of the markers and the repeat counter. This requirement will be possible in logarithmic space. Therefore, $BOTH_{NFA} \in NL$. Next user need to prove that $BOTH_{NFA}$ is also NL-Hard.

Comment

**Step 2** of 3

Now, consider a log space reduction from some language $A$ in $NL$ to $BOTH_{NFA}$. For this purpose, consider a Nondeterministic TM $M$ which decides $A$ in $O(\log n)$.

• Given an input string $\omega$, construct two different NFAs $M_1$ and $M_2$ in log space which accepts $\omega$ if and only if $M$ accepts $\omega$.

• The states of $M_1$ and $M_2$ are the configurations of $M$ on $\omega$. For configuration $q_1$ and $q_2$ of $M$ on $\omega$, the pair $(q_1, q_2)$ are two states both in $M_1$ and $M_2$ if $q_2$ is the possible next configuration of $M$ starting from $q_1$.

• The above statement shows, if $M$'s transition function follows that state $q_1$ together with its state symbols under corresponding input symbol.

• Also, each of the work tape heads can find the next state and also the head actions to take from $q_1$ to $q_2$, then $q_1$ and $q_2$ are two states in both $M_1$ ands $M_2$.

This configuration reduces $A$ to $BOTH_{NFA}$ because whenever $M$ accepts an input, some its computational branch accepts some state transition in both of the machines $M_1$ and $M_2$.

Comments (1)

**Step 3** of 3

Again conversely, if $M_1$ and $M_2$ accepts a string $\omega$ some computational branch follows in $M$ and accepts $\omega$. Now to show that the given reduction works in log space, consider the log space transducer that outputs $\langle M_1, M_2\rangle$ on input $\omega$.

• Consider describing $M_1$ and $M_2$ by listing their states. Here each state is a configuration of $M$ on input of $\omega$ and it can be denoted by $c \log n$ space where $c$ is some constant.

• Now, it can be said, transducer attends via all possible strings of length $c \log n$ sequentially and checks if each one is a legal configuration of $M$ on $\omega$ and outputs only those which passes the checking.

• Hence, it is sufficient to verify in log space that any configuration $q_1$ of $M$ on input $\omega$ can find another configuration $q_2$. This is because, the transducer only needs to check for the actual tape contents under the each head locations given in $q_1$ and used to determine that $M$'s transition function would produce $q_2$ as a result.

• The transducer verifies all pairs $(q_1, q_2)$ in each turn to find the states in $M_1$ and $M_2$. **Hence it is proved that** $BOTH_{NFA}$ **is** $NL-Complete$.

Comment

## Problem

Show that $A_{NFA}$ is NL-complete.

## Step-by-step solution

**Step 1** of 1

Consider

$A_{NFA} = \{\langle M, w\rangle \mid M \text{ is } NFA \text{ and } M \text{ accepts } w\}$ and
$\text{PATH}=\{\langle G,s,t\rangle \mid G \text{ is any directed graph also } G \text{ has path from } s \text{ to } t\}$

Firstly it is required to show $A_{NFA}$ is $NL$.

Use a verifier /certificate definition of $NL$, similar to verifier /certificate of $NP$. $A_{NFA}$ is in $NL$ if there is some deterministic type Turing machine $V$ such that $V$ uses $O(\log n)$ space and $\langle M, w\rangle$ in $A_{NFA}$ iff there is any certificate $c$ such that $V$ accepts $\langle\langle M, w\rangle, c\rangle$. The certificate will be a path in the underlying graph of $NFA$ $M$, and the verifier operates as given below:

- The states in the path denoted by $(p_0, \ldots\ldots, p_n)$.
- Check that the $p_0 = q_0$.
- For each $i \in \{1, \ldots., n\}$, check that $\delta(p_{i-1}, w_i) = p_i$.
- Check that $p_n$ is accepting state.
- If any checks fail, then reject. Otherwise accept.

This algorithm checks every condition which is required by the definition of "$M$ accept $w$", if it has $M$ accept $w$ iff $V$ accept $\langle\langle M, w\rangle, (p_0, \ldots, p_n)\rangle$ for any path $(p_0, \ldots, p_n)$.Therefore it is concluded that $A_{NFA}$ is in $NL$.

Now it is required to prove that $PATH \leq_M^L A_{NFA}$. Since Path is $NL-complete$, this shows that $A_{NFA}$ is $NL-complete$.

The reduction of mapping $\langle G,s,t\rangle \to \langle M,\varepsilon\rangle$, where $M$ is $NFA$ whose graph is $G$ with $\in$ labels on each edge, where $s$ is initial state, and $t$ is its accept state. If $G$ has path from $s$ to $t$, then there is few sequence of the vertices $(v_1, v_2, v_3, \ldots., v_k)$ so that

- $v_1 = s$
- $(v_i, v_{i+1}) \in E(G)$ for each $i \in \{1, 2, 3 \ldots., k-1\}$
- $v_k = t$

Thus, $(v_1, v_2, v_3, \ldots, v_k)$ is sequence of the state in $M$ such that

- $v_1$ is initial state,
- $\delta(v_i \in) = v_{i+1}$ for each $i \in \{1, 2, 3, \ldots, k-1\}$
- $v_k$ is the terminal state,

Three conditions which are necessary to conclude that $M$ accepts $\varepsilon$. Conversely, if $G$ has not any path from point $s$ to point $t$, then the path will violate one of three given conditions for having the path, so any state sequence will violate any one of three conditions to accept. Therefore $PATH \leq_M^L A_{NFA}$ (and thus $A_{NFA}$ is $NL-complete$).

Comment

## Problem

Show that $E_{DFA}$ is NL-complete.

## Step-by-step solution

**Step 1** of 1

• $E_{DFA}$ is in $co-NL$ (a sufficient certificate is an accepted string-one always exists of length less than the number of states), thus it is in $NL$. Now it is required to prove that $NL-\text{hardness}$ by a reduction from $\overline{PATH}$ (which is $co-NL-\text{complete}$, and thus $NL-\text{complete}$).

• The idea of the reduction from $\overline{PATH}$ is simple. Given $G=(V,E)$ and vertices $s,t$ and will construct a $DFA$ having state graph is $G$, $s$ is the initial state and $t$ is the final state. Then the language of this $DFA$ is empty if and only if $t$ is not reachable from $s$ $\left(\text{that is,}\langle G,s,t\rangle \notin \overline{PATH}\right)$.

• So, it sufficient to show this is possible with a log-space transducer.

• First, for each vertex, count the maximum out-degree $d$. Our alphabet will have $|\Sigma|=d$ (in particular, will take $\Sigma=\{1,\ldots,d\}$ where one can write each number in $\log d=O(\log m)$ bits). This computation is easily done in log-space, by counting at each vertex and only maintaining a maximum value.

• Each vertex $u$ will be translated (in order vertices appear on the tape) into a state, and each edge $(u,v_i)$ (in the order the edges out of $u$ appear on the tape) into a transition rule $\delta(u,i)=v_i$.

• If reach the end of the list of vertices without giving transitions for all $d$ letters, add copies of the last edge visited so that $u$ has transitions for all letters. This step is done in log-space, since at most one vertex and two edges at a time.

• It is then easy to set $s$ to be the start state, and $t$ to be the final state. So it is possible to do this reduction in log-space, and hence, $E_{DFA}$ is NL-complete.

Comment

## Problem

Show that *2SAT* is NL-complete.

## Step-by-step solution

**Step 1** of 4

<u>NL – completeness:-</u> A language *B* is *NL*- complete if

1. $B \in NL$, and

2. Every *A* in *NL* is log space reducible to *B,* that is *B* is $NL-$hard.

Now we have to prove that 2*SAT* is *NL* –complete.

We know that $2SAT = \{ \langle \phi \rangle \mid \phi$ is a satisfiable Boolean formula in $2cnf \}$.

We know the fact that $NL = CONL$

Thus if we prove that $\overline{2SAT}$ is $NL-$complete, then that implies that 2*SAT* is $NL-$complete.

Comment

**Step 2** of 4

To prove $\overline{2SAT}$ is *NL*- complete, we have to prove the two conditions of $NL-$completeness.

(i) $\overline{2SAT} \in NL$:

In order to prove $\overline{2SAT} \in NL$ construct the graph $NTM$ $M$ as follows:

$M =$ "On input $\phi$:

(1) Construct graph *G* such that $\phi$ is not satisfiable if and only if there two vertices $u, v$

the paths $uv$ and $vu$ are in *G*.

(2) For each variable *x* create a node labeled *x* and another labeled $\bar{x}$.

(3) Now non-deterministically select a vertex *x* in *G*.

(4) Check if *G* contain both *x* $x\bar{x}$ and $\bar{x}x$ paths using $NL-$algorithm for $PATH$.

(5) If both paths exist, then accept

(6) Otherwise reject ".

The entire construction is done in log – space and *M* decides $\overline{2SAT}$ in log space only.

Thus $\overline{2SAT} \in NL$

Comment

**Step 3** of 4

(ii) $\underline{\overline{2SAT} \text{ is } NL_\text{hard:}}$

We do this by reducing path to $\overline{2SAT}$.

<u>Construction of</u> $\phi$:

• Given $G, s, t$ we have to construct a $2-cnf$ formula $\phi$ such that *G* has an $s,t-$path if and only if $\phi$ is unsatisfiable.

• Let $v(G) = \{s, t, y_1, \ldots y_n\}$

• The resulting formula $\phi$ consists of $m+1$ variables $x, y_1, \ldots y_m$

• The vertex *s* is identified with *x* and the vertex *t* with $\bar{x}$.

• The ıasses of $\phi$ will be $x \vee x$ and $\bar{u} \vee u$ every edge $(u,v)$ of *G*.

Comment

**Step 4** of 4

→ In order to show that this construction works, first assume that *G* has an $s,t$ path. Let this path be $s,u_1,...u_k t$. Then $\phi$ contains the clause $(\bar{x} \vee u_1),(\bar{u}_1 \vee u_2)...(\bar{u}_k,\bar{x})$.

If we designs $x=1$, at least one of these clauses is not satisfied on the other hand, if $x=0$, then the clause $(x \vee x)$ of $\phi$ is not satisfied. So $\phi$ is zero for any assignment of $x,y_1,...y_m$.

→ Conversely, suppose that *G* contains no $s-t$ path.

• Let *U* be the set of vertices in *G* reachable from $s,V$ the set of vertices from which *t* is reachable and $w:V(G)(U \vee V)$.

By hypothesis and construction $U,V,W$ are pair wise disjoint, and there are no edges from *U* to $V \cup W$ or from $U \cup W$ to V. let us assign the rue value to every variable in $U \cup W$ and the false value to every variable in *V*. it is now simple to check that $\phi$ is satisfied for this truth assignment.

Comment

## Problem

Let $CNF_{H1}$ = { $\langle \phi \rangle$ I ? is a satisfiable cnf-formula where each clause contains any number of positive literals and at most one negated literal. Furthermore, each negated literal has at most one occurrence in ?}. Show that $CNF_{H1}$ is NL- complete.

## Step-by-step solution

**Step 1** of 3

Consider the $\mathrm{CNF_{H1}}$, which is defined as:

$$\mathrm{CNF_{H1}} = \{<\varnothing>|\ \varnothing \text{ is a satisfiable cnf-formula where each clause contain nay number of positive literals and at most one negative literals}\}$$

Now, it is known that $CNF_{H1} \in NL$. It can be proved by using the following approach: Situation is quite clear as all user needs to find the d**ependability** of $\mathrm{CNF_{H1}}$ on $\mathrm{P}$ and $\mathrm{NL}$.

Comment

**Step 2** of 3

**Dependability can be viewed as:**

1. Consider that $\mathrm{CNF_2} \in NL$

Consider first situation of $\phi$ and consider it belongs to some x and if there is any $\neg x$ then reject.

Consider one more situation of $\phi$ that will be belonging to some other variable let's say $\mathrm{y}$ and here situation will rejected if there is any $\neg y$ otherwise accepted.

• **Two situations are discussed here remove those situation from $\phi$ and relate those two situations, let's say M and N, to $\phi$ and call the result**. This way it is proved that $\mathbf{CNF_2} \in NL$.

Comment

**Step 3** of 3

2. Consider another situation $\mathrm{CNF_3} \in NL$

Consider first situation of $\phi$ and consider it belongs to some p and if there is any $\neg p$ then reject.

Consider one more situation of $\phi$ that will be belonging to some other variable, let's say $\mathrm{q}$ and $\mathrm{r}$ here situation will rejected if there is any $\neg q$ otherwise accepted.

• **Two situations are discussed here remove those situation from $\phi$ and relate those two situations, let's say A and B, to $\phi$ and call the result.** This way it is proved that $\mathbf{CNF_3} \in NL$.

Similarly**, this situation will be held true for $\mathbf{CNF_{H1}}$**. In other words**, it can be said that**" $CNF_{H1} \in NL$ " **or** $\mathbf{CNF_{H1}}$ is $NL-complete$.

Comment

## Problem

Give an example of an NL-complete context-free language.

## Step-by-step solution

**Step 1** of 3

The statement of Ginsburg theorem state that, "Suppose $G$ is used to generate a language $L(G)$ where $G$ is defined as a reducible context free grammar. Then $|L(G)\cap\{0,1\}^n|$ will show a polynomial behavior in $n$ if and only if for each non-terminal $z, l(z)$ and $r(z)$ are commutative".

Comment

**Step 2** of 3

Now, suppose $M\subseteq\{0,1\}^*$ be a language which **is context free and exponential** in size. The above context free language shows a $NP-complete$ behavior.

• To perform this, suppose, $D$ is defined as **a reduced-free grammar** for $M$. Here, $M_n$ will not be in polynomial of n because $M$ consists an **exponential size**.

• Now from the above given theorem, it can be said that there exists a nonterminal and for each non-terminal $Z, l(Z)$ and $r(Z)$ are **commutative**.

Comment

**Step 3** of 3

Consider, $l(Z)$ (where, $a_1, a_2 \in l(Z)$) is not showing the commutative property and $a_1, a_2 \neq a_2, a_1$. Hence, there exists a position $k$ such that $(a_1, a_2)_k = 0$ and $(a_2, a_1)_k = 1$.

• As from the above explanation, there exists $(b_1, b_2)\in\{0,1\}^*$ in such a way that $Z \succ^* a_1Zb_1$ and $Z \succ^* a_2Zb_2$. An arbitrary $1s$ and $0s$ can be generated at the position $k\cdot|a_1a_2|+i$ for any $k$, by applying either $Z \succ^* a_1a_2Zb_2b_1$ or $Z \succ^* a_2a_1Zb_1b_2$, $k$ times.

• Now, to reach $Z$ from $M_0$, user can use $M_0 \succ^* aZb$. Again, to acquire a word in $\{0,1\}^*$ for some $x, y, w \subseteq \{0,1\}^*$, $Z \succ^* w$ can be used.

• Hence from the above discussion, $I := \{|x|+k\cdot|a_1a_2|+i : 0\le k\le n-1\}$ is of size $n$ can be obtained if $N$ is set as $N := \{|x|+|y|+|w|+n(|a_1a_2|+|b_1b_2|)\}$ and also $I$ can be shattered by $M_N$.

• All the calculations (which are done above) take a time in $O(n)$ and $N$ is linier in $n$ and it is already known that $S-SAT$ is $NP-hard$ and $S-SAT \in NP$. Hence, it shows $NP-complete$ behavior.

**Hence the above explanation shows, the context free language $M\subseteq\{0,1\}^*$ is $NP-complete$.**

Comment

## Problem

Define *CYCLE* = { $\langle G \rangle$ I G is a directed graph that contains a directed cycle}. Show that *CYCLE* is NL-complete.

## Step-by-step solution

**Step 1** of 2

<u>NL – completeness:</u>

A language *B* is *NL*- complete if

1. $B \in NL$, and

2. Every *A* in *NL* is log space reducible to *B*.

Given that

$CYCLE = \{ \langle G \rangle \mid G$ is a directed graph that contains a directed cycle}.

Now we have to prove that CYCLE is *NL* – complete.

1. $\underline{CYCLE \in NL}$:

We know that

"*NL* is the class of languages that are decidable in logarithmic space on a Non deterministic Turing machine"

Let *N* be the nondeterministic Turing machine that decides $CYCEL$.

The construction of *N* is as follows:

$N =$ "On input $\langle G \rangle$ ( *G* is an directed graph):

1. Select a vertex *u* as starting vertex.

2. Select an edge $(u,v)$ from *u*.

3. Run $PATH(u,v)$.

4. Start traversal through $(u,v)$, if we come back to *u* through on edge different that

$(u,v)$, then direct cycle will exist

5. Otherwise, reject."

• Since all vertices and all the edges are enumerated in log space *N* decides CYCLE in logarithmic space.

• Therefore, $CYCLE \in NL$.

Comment

**Step 2** of 2

2. $\underline{PATH \leq_L CYCLE}$:

• Now we have to reduce $PATH$ to $CYCLE$.

• For that we have modify the *PATH* problem instance $\langle G,s,t \rangle$ by adding an edge from *t* to *s* in *G*.

• If path exists from *s* to *t* in *G* then direct cycle will exist in modified *G*.

• But some cycles may already be present in *G*.

• So, so solve that problem change *G* so that it contains no cycles.

• A leveled directed graph is non where the nodes are divided into graphs, $A_1, A_2, \ldots A_k$ called levels.

• Only edges from one level to next higher level are permitted.

• $G'$ is the leveled graph of *G* which has two nodes *s* and *t*, and *m* nodes in total.

• Draw an edge from node *i* at each level to node, *j* in the next level if *G* contain an edge from *i* to *j*.

• Also, Draw on edge from node 1 in each level to node *l* in the next level.

• Let $s'$ be the node *s* in the first level and $t'$ be the node *t* in the next level.

• Graph *G* contains a path from *s* to *t* iff $G'$ contain a path from $s'$ to $t'$.

• If we add an edge from $t'$ to $s'$ in $G'$ then reduction from *PATH* to *CYCLE* will be obtained, this reduction is implemented in log space.

Therefore (1) and (2) $CYCEL$ is $NL-$complete.

Comment

## Problem

Prove that TIME($2^n$) = TIME($2^{n+1}$).

## Step-by-step solution

**Step 1** of 1

**Proof of** $\mathbf{TIME}\left(2^n\right)=\mathbf{TIME}\left(2^{n+1}\right)$

Classification of computation time can be done on the basis of the minimum time required by the most efficient algorithm to determine upper and lower bounds.

Usually Big-O notation is used to state the lower and upper bond which **hides smaller terms** and **constant factors.**

• Big-O notation is used to define the **time complexity** classes. So there is no effect of constant factors used in complexity calculation.

• Therefore the time complexity of the function $2^{n+1}$ in big-O notation is O $\left(2^n\right)$ .

• Thus, $\mathbf{A}\in\mathbf{TIME}\left(2^n\right)$ if and only if $\mathbf{A}\in\mathbf{TIME}\left(2^{n+1}\right)$.

• Hence, $\mathbf{TIME}\left(2^n\right)=\mathbf{TIME}\left(2^{n+1}\right)$

Comment

## Problem

Prove that TIME($2^n$) $\subsetneq$ TIME($2^{2n}$).

## Step-by-step solution

**Step 1** of 1

**Proving** $\mathbf{TIME}(2^n) \subsetneq \mathbf{TIME}(2^{2n})$

The containment $\mathrm{TIME}(2^n) \subseteq \mathrm{TIME}(2^{2n})$ holds because $2^n \le 2^{2n}$. Now, consider the **time hierarchy theorem** which says that, "if f, g are time-constructible functions and $f(n)\log f(n) = O(g(n))$ then $\mathrm{DTIME}(f(n)) \subsetneq D\mathrm{TIME}(g(n))$.

**Thus, the above containment is proper by virtue of the time hierarchy theorem as discussed**.

• A Turing machine can write the number 1 followed by 2n 0s in $O(2^{2n})$ time. So, the function $2^{2n}$ is time constructible

• Hence, the time hierarchy theorem guarantees that a language A exists that can be decided in $O(2^{2n})$ time but not in $o(2^{2n}/\log 2^{2n}) = o(2^{2n}/2n)$ time.

• Therefore, $A \in TIME(2^{2n})$ but $A \notin TIME(2^n)$.

**From the above explanation it can be said that** $\mathrm{TIME}(2^n) \subsetneq \mathrm{TIME}(2^{2n})$ .

Comment

## Problem

Prove that NTIME(n) $\subsetneq$ PSPACE.

## Step-by-step solution

**Step 1** of 1

$\mathbf{NTIME(n)}$ **is strict subset of** $\mathbf{PSPACE(n)}$

At most $t(n)$ tape cells on each branch can be used by any Turing machine that operates in time $t(n)$ on each computation branch. So, it can be stated that $\mathrm{NTIME}(n) \subsetneq \mathrm{NSPACE}(n)$

• Now, consider the **Savitch's theorem** which says that: "Let $f : N \to R$ be a function, with $f(n) \geq n$ then $\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}\left((f(n))^2\right)$".

Therefore according to Savitch's theorem $\mathrm{NSPACE}(n) \subseteq \mathrm{SPACE}(n^2)$.

• Now, consider the **space hierarchy theorem** which says that "if $g$ is space-constructible ( $1^n \to 1^{g(n)}$ can be computed in space $O(g(n))$), $f(n) = O(g(n))$ then $\mathrm{SPACE}(f(n)) \subsetneq \mathrm{SPACE}(g(n))$.

Therefore, according to space hierarchy theorem it can be said that $\mathrm{SPACE}(n^2) \subsetneq \mathrm{SPACE}(n^3)$. The result follows because $\mathrm{SPACE}(n^3) \subseteq \mathrm{PSPACE}$.

**From the above explanation, it can be said that** $\mathrm{NTIME}(n) \subsetneq \mathrm{PSPACE}(n)$.

Comment

**Problem**

Show how the circuit depicted in Figure 9.26 computes on input 0110 by showing the values computed by all of the gates, as we did in Figure 9.24.

Figure 9.26

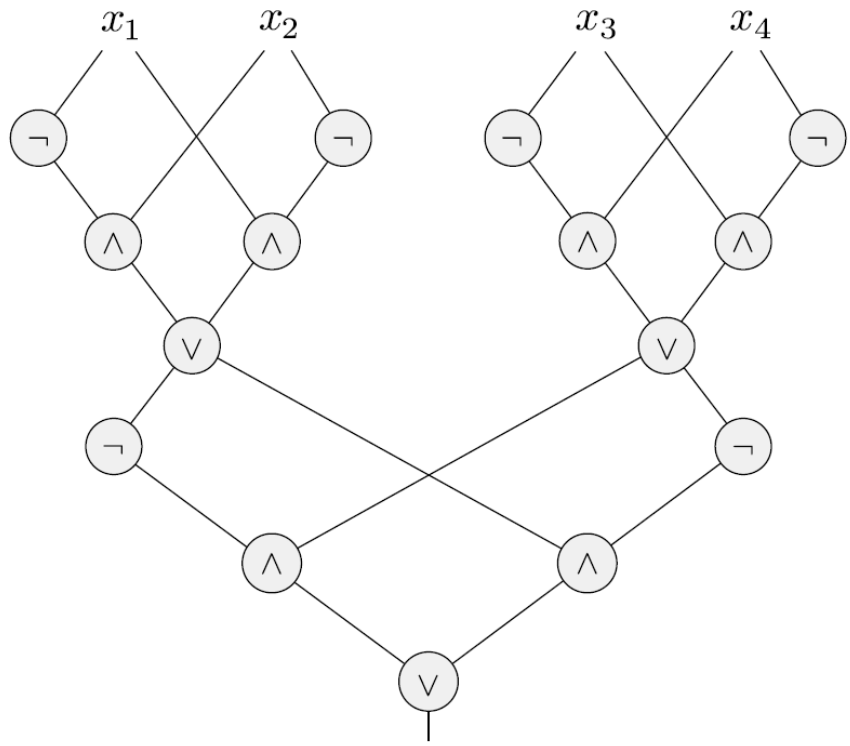



**FIGURE 9.26**
A Boolean circuit that computes the parity function on 4 variables

Figure 9.24

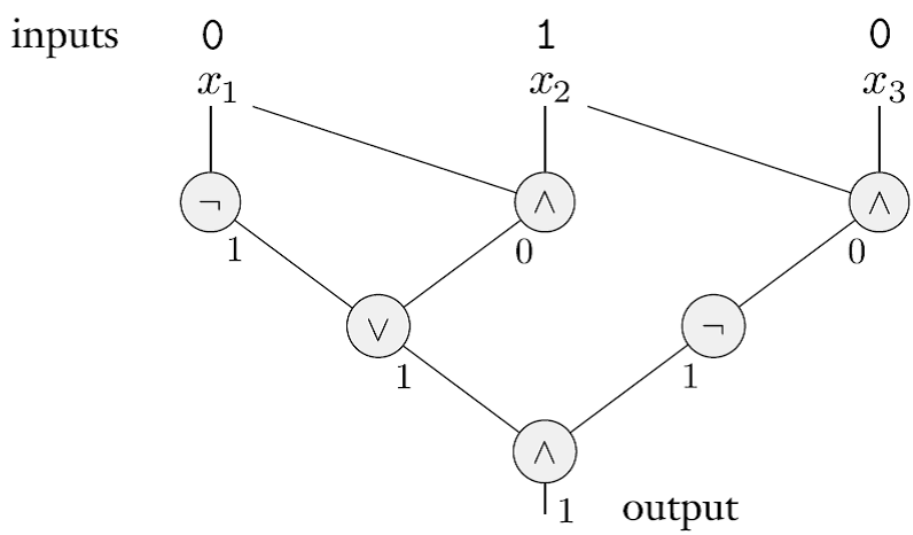



**FIGURE 9.24**
An example of a Boolean circuit computing

## Step-by-step solution

**Step 1** of 6

**Representation of the parity functional circuit**

The *n*-input *parity* function $parity_n : \{0,1\}^n \to \{0,1\}$ gives an output 1 if an odd number of 1's appear in the input variable and otherwise gives 0. The circuit depicted here (refer figure 9.26) is used as a parity functional circuit on four variable.

In this parity functional circuit, a number of gates are used. Therefore, first of all, the working of gates will be discussed here to understand the process performed in that circuit.

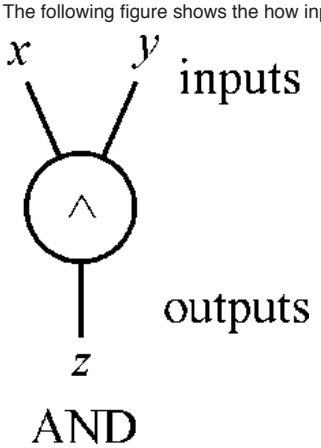
Comment

**Step 2** of 6

The Truth table for and operator is as follows:

| **Input** | Output z |
|---|---|
| x | y |
| 0 | 0 |
| 1 | 0 |
| 0 | 1 |
| 1 | 1 |

The following figure shows the how input and output can be taken in the above table (for and operator):



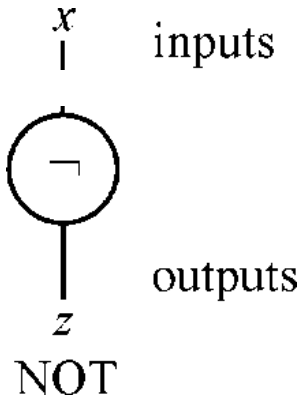

Comment

**Step 3** of 6

The Truth table for not operator is as follow:

| Input(x) | Output(z) |
|---|---|
| 0 | 1 |
| 1 | 0 |

The following figure shows the how input and output can be taken in the above table (for not operator):



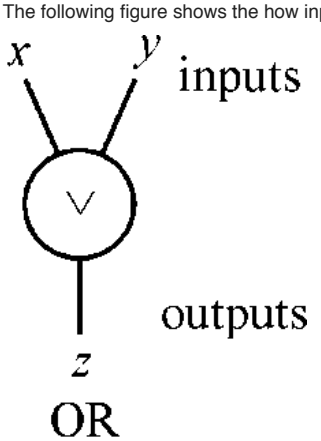

Comment

**Step 4** of 6

The Truth table for or operator is as follow:

| Input | Output z | |
|---|---|---|
| x | y | |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

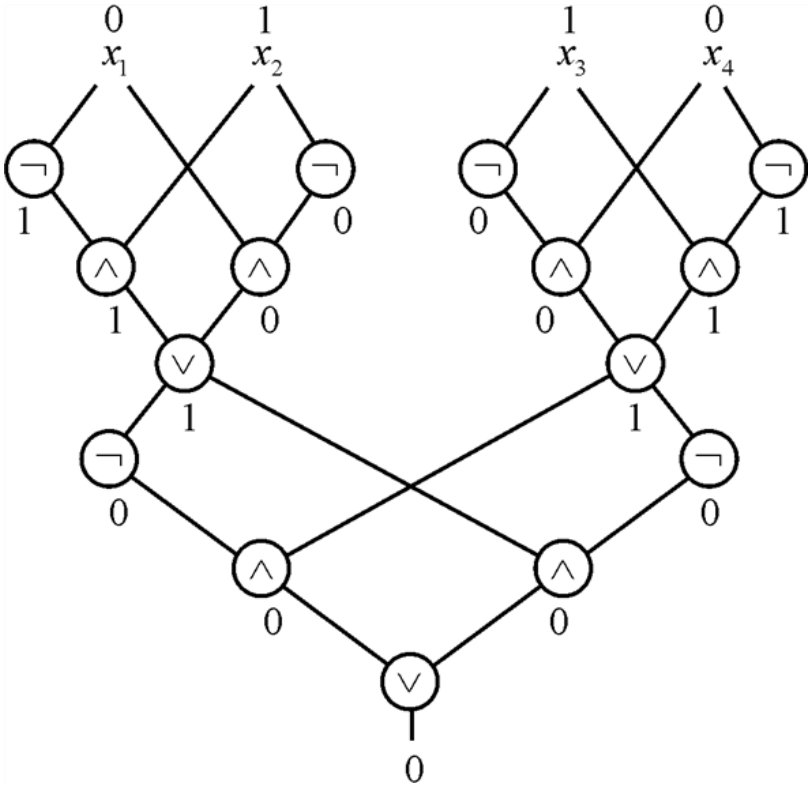
The following figure shows the how input and output can be taken in the above table (for or operator):




Comment

**Step 5** of 6

**Consider the figure given below:**




**Above figure shows the functional parity circuit on 4 variables**. Here 0110 is taken as an input variable. Now AND gate, OR gate and NOT gate applied on the input in the

Comment

**Step 6** of 6

same manner as it is discussed above. The final output is 0 because here even number of 1's has taken (as the definition of parity function says).

Comment

## Problem

Give a circuit that computes the parity function on three input variables and show how it computes on input 011.

## Step-by-step solution

**Step 1** of 7

The *n*-input **parity function** is the Boolean function $f : \{0,1\}^n \to \{0,1\}$ gives an output 1 if an odd number of 1's appear in the input variable and otherwise gives 0. Here $f$ is defined as:

$f(x) = x_1 \oplus x_2 \oplus x_3 \oplus ... \oplus x_n$ , where $\oplus$ denotes an EX-OR gate operation.

• In this parity functional circuit, a number of gates are used. Therefore, first of all, the working of gates will be discussed here to understand the process performed in that circuit.

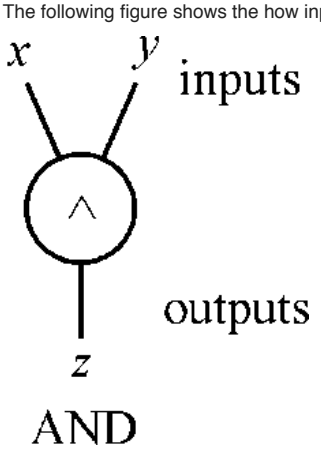
Comment

**Step 2** of 7

The Truth table for **and** operator is as follow:

| **Input** | Output z | |
|---|---|---|
| x | y | |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

The following figure shows the how input and output can be taken in the above table (for **and** operator):




Comment

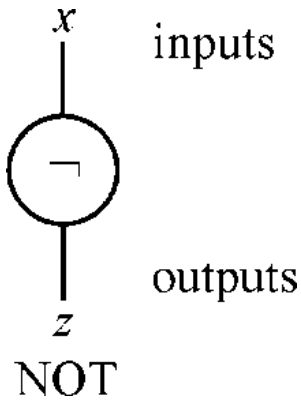
**Step 3** of 7

The Truth table for **not** operator is as follow:

| Input(x) | Output(z) |
|---|---|

| | |
|---|---|
| 0<br>1 | 1<br>0 |

The following figure shows the how input and output can be taken in the above table (for **not** operator):



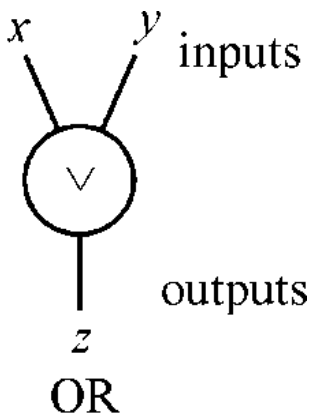

Comment

**Step 4** of 7

The Truth table for **or** operator is as follow:

| Input | Output<br>z |
|---|---|
| x | y |
| 0<br>0<br>1<br>1 | 0<br>1<br>0<br>1 | 0<br>1<br>1<br>1 |

The following figure shows the how input and output can be taken in the above table (for **or** operator):




Comment

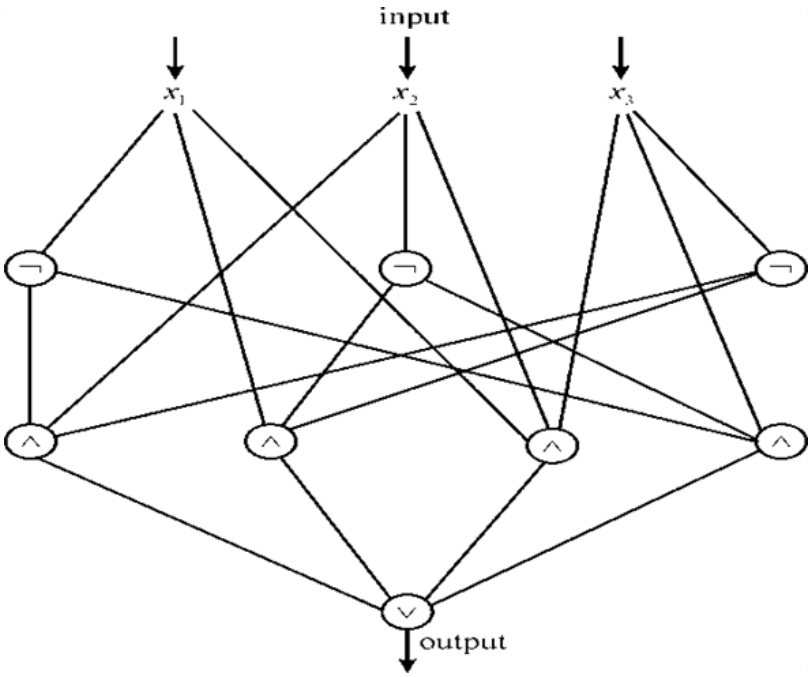
**Step 5** of 7

From the definition of **parity function**, the n-input parity functional circuit can be obtained by applying an EX-OR gate operation on the n- inputs.

• For three input variables, the parity function may be defined as:

$$
\begin{aligned}
f(x) &= x_1 \oplus x_2 \oplus x_3 \\
&= \left[ (x_1 \cdot \overline{x}_2) + (\overline{x}_1 \cdot x_2) \right] \oplus x_3 \\
&= \left[ x_1 \cdot \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_2 \cdot \overline{x}_3 + x_1 \cdot x_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot \overline{x}_2 \cdot x_3 \right]
\end{aligned}
$$

Consider the figure which is given below:




•

Comment

**Step 6** of 7

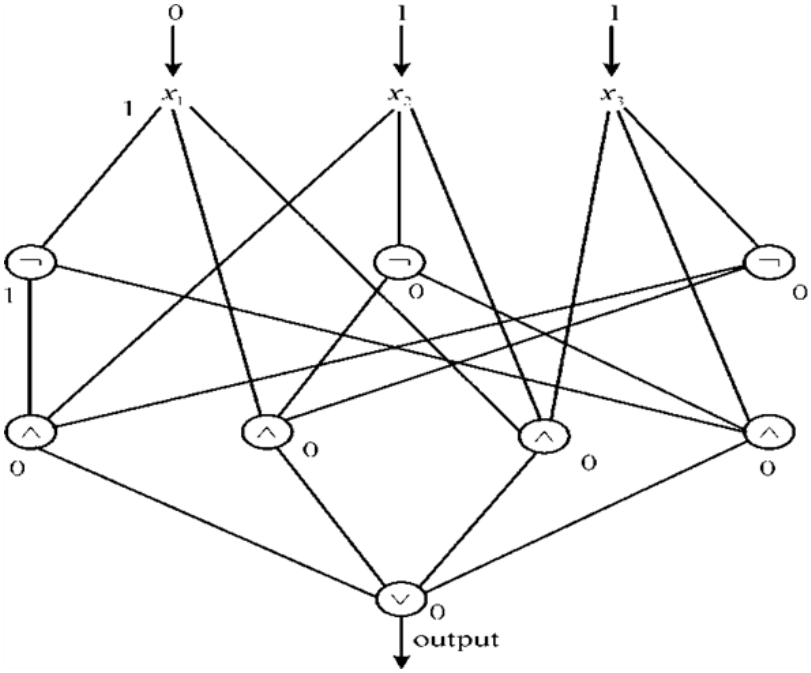
Above figure shows the functional circuit on three input variable. It shows how three variables are taken as an input, and how an EX-OR gate operation is applied on these inputs.

• An EX-OR gate consists of an AND, NOT and OR gate operations.

Comment

**Step 7** of 7

Consider the figure which is given below:




Above figure shows, the computation performed by three input parity functional circuit when 001 is taken as an input. After taking an input , an EX-OR operation is applied as discussed above.

• The output given by the parity functional circuit is 0 because even number of 1's are present in the input.

• which follow the definition of parity funtional circuit(that is, it gives an output 1 if an odd number of 1's appear in the input variable and otherwise gives 0).

Comment

## Problem

Prove that if A ? P, then $P^A$ = P.

## Step-by-step solution

**Step 1** of 1

**Given:**

Language *A* belongs to the polynomial type oracle Turing machine *P*.

$A \in P$

**Proof:**

$P^A$, is basically the class of the language which is decidable with the polynomial Turing machine which uses language *A*.

Here, it is to be proved that $P^A = P$

**Construction:**

Language *A* is the device which checks whether the string *w* is the member of *A*.

Suppose string *w* contains the value $\{a^n b^n\}$ and user has given the string $\{aabb\}$ then it is accepted by *A*. If the string $\{aabbb\}$ is specified then it is not expected by the language *A*.

Now, user checks this for the poly-time Turing machine:

Suppose *N* is the poly-time Turing machine, here, construction of the machine is done so that it can be proved that $P^A$, is basically the class of the language which is decidable with the polynomial Turing machine which uses language *A*.

• Computation path for *N* is done and it is found that it leads to the acceptance of the polynomial.

• If the string is accepted then yes query will execute if it is not accepted then no query will execute.

• Machine *N* checks whether all the string is accepted and rejected

• If the guess is right then string is accepted.

Here, for all the string computation is done and hence it is proved that $P^A = P$

Comment

## Problem

Give regular expressions with exponentiation that generate the following languages over the alphabet {0,1}.

A**a.** All strings of length 500

A**b.** All strings of length 500 or less

A**c.** All strings of length 500 or more

A**d.** All strings of length different than 500

**e.** All strings that contain exactly 500 1s

**f.** All strings that contain at least 500 1s

**g.** All strings that contain at most 500 1s

**h.** All strings of length 500 or more that contain a 0 in the 500th position

**i.** All strings that contain two 0s that have at least 500 symbols between them

## Step-by-step solution

**Step 1** of 8

**Regular expressions:**

Regular expression are the expression by which searching is done very easily. It is used for matching the combination of character in the string.

Regular expression consists of normal characters or the combination of the special character.

Power of an alphabet is donated by $\Sigma^k$. It is basically the combination of the string of the power length which is k and the power is donated by the symbol $\Sigma$.

**a) Given:**

Every string is of the length 500

**Regular expression with the exponentiation:**

$$\Sigma^{500}$$

• Power of an alphabet is donated by $\Sigma^k$. Here, as the string is of length 500 so value of *k* is assigned to 500.

• It represents the combination of the 500 string with exponentiation.

Comment

**Step 2** of 8

**b) Given:**

Every string is of length 500 or it is less than 500

**Regular expression with the exponentiation:**

$$(\Sigma \cup \varepsilon)^{500}$$

• In the above regular expression $\varepsilon$ is donated by the empty string. The union of Exponentiation and $\varepsilon$ will be less than or equal to 500.

• If there is an empty string then combination will of less than 500 strings. If there is no empty string then the combination will be equal to 500.

Comment

**Step 3** of 8

c) **Given:**

Every string is of length 500 or it is more than 500

**Regular expression with the exponentiation:**

$$\Sigma^{500}\Sigma^{*}$$

• In the above regular expression is $\Sigma^*$ donates all the string which is more than 500. $\Sigma^{500}$ donate the string of the length 500.

• The multiplication of both the exponentiation will result the string of the length 500 or more than 500.

Comment

**Step 4** of 8

d) **Given:**

Every string should have length different than 500.

**Regular expression with the exponentiation:**

$$(\Sigma \cup \varepsilon)^{499} \cup \Sigma^{501}\Sigma^*$$

• In the above regular expression is $(\Sigma \cup \varepsilon)^{499}$ donates all the string which is 499 or less than 499 as $\varepsilon$ donate an empty string.

• Whereas the regular expression $\Sigma^{501}\Sigma^*$ donates all the string which is equal to 501 and more than 501.

• The union of the both the regular expression gives the string either greater than 500 or less than 500 but not the string of length 500.

Comment

**Step 5** of 8

e) **Given:**

String containing total 500, 1's exactly.

**Regular expression with the exponentiation:**

$(0)^n(1)^{500}$, where $n \geq 0$

• In the above regular expression is $(0)^n(1)^{500}$ donates all the string which contains exactly 500 1's and number of 0's in the string are greater than or equal to 0.

Comment

**Step 6** of 8

f) **Given:**

String contains at least 500, 1's.

**Regular expression with the exponentiation:**

$0^n 1^{500} 1^*$, where $n \geq 0$

• In the above regular expression is $0^n 1^{500} 1^*$ donates all the string which contains 500 1's or more than 500 1's and number of 0's in the string are greater than or equal to 0.

Comment

**Step 7** of 8

g) **Given:**

String containing maximum 500 1's

**Regular expression with the exponentiation:**

$0^n(1 \cup \varepsilon)^{500}$, where $n \geq 0$

• In the above regular expression is $0^n(1 \cup \varepsilon)^{500}$ donates all the string which contains less than or equal to 500 1's and number of 0's in the string are greater than or equal to 0.

• If there is no empty string then the combination will be equal to 500.

Comment

**Step 8** of 8

h) **Given:**

String of minimum 500 lengths and the position of 500th position are fixed with 0

**Regular expression with the exponentiation:**

$$(\Sigma)^{499}(0)(\Sigma)^{*}$$

• In the above regular expression $\Sigma^{*}$ donates all the string which is more than 500. $\Sigma^{499}$ donate the string of the length 499. 0 is present after writing all the string of length 499 which implies 0 is present in the 500 position.

**i) Given:**

String containing minimum 500 symbols between two 0's

**Regular expression with the exponentiation:**

$$(0)(\Sigma^{500})(\Sigma)^{*}(0)$$

• In the above regular expression there are minimum 500 1's between two 0's.

• $\Sigma^{*}$ donate all the string which is more than 500.

Comment

## Problem

If R is a regular expression, let $R^{\{m,n\}}$ represent the expression

$$R^m \cup R^{m+1} \cup \cdots \cup R^n.$$

Show how to implement the $R^{\{m,n\}}$ operator, using the ordinary exponentiation operator, but without "· · · ".

## Step-by-step solution

**Step 1** of 1

Consider the following Regular expression:

$$R^m \cup R^{m+1} \cup ... \cup R^n$$

Here, user needs to be proved the implementation of $R^{m,n}$ operator with the help of the ordinary operator but without using "…".

This can be proved with the help of computation history.

Computation history is basically the series of configuration that is done by the machine at the time of processing the input.

Turing machine *M* is being considered. If the machine does not halt on the particular input then acceptation and rejection of the string is not done by the Turing machine.

With the help of deterministic machines computation history can be calculated for the particular input.

Thus it can be said that all the states which are there in the Turing machine is basically countable and even the arrangement of the all the symbol present in the tape are countable.

So, the union of the two set in the Turing machine is also said to be countable.

As, it is being proved that all the sequence is countable so exponentiation of this sequence can be written:

$$R^{m,n} = \sum_{m=0}^{n} \cup (A)^n$$

Comment

## Problem

Show that if NP = $P^{SAT}$, then NP = coNP.

## Step-by-step solution

**Step 1** of 1

If $NP = P^{SAT}$ is assumed then, there is only need to show that $NP = coNP$. It can be conclude directly from the assumption $NP = P^{SAT}$, that $P^{SAT} \subseteq NP$. As $coNP \subseteq P^{SAT}$ is already known, which result in $coNP \subseteq NP$.

• Now it is known that $P$ is closed under the complement operation, so is $P^{SAT}$, because it can be just swap the reject and accept states. It can be concluded that:

$L \in P^{SAT} \Rightarrow \overline{L} \in P^{SAT}$.

• The given statement can be managed by using the prediction $NP = P^{SAT}$, because if this would be the case any language in $P^{SAT}$ would be in $NP$ and vice versa.

• For which $NP$ also has to be closed under the complement operation $L \in NP \Rightarrow \overline{L} \in NP$, that is just the same as $L \in NP \Rightarrow L \in coNP$ from the definition of $coNP$.

• In other words, from the above explanation it can be said that $NP \subseteq coNP$. Hence $NP = coNP$.

Comment

## Problem

Problem 8.13 showed that $A_{LBA}$ is PSPACE-complete.

**a.** Do we know whether $A_{LBA}$ ? NL? Explain your answer.

**b.** Do we know whether $A_{LBA}$ ? P? Explain your answer.

## Step-by-step solution

**Step 1** of 3

It is being given and it is being proved in the previous chapter that $A_{LBA} \in PSPACE$.

*PSPACE* is basically the class of the language, whether the language belongs to deterministic finite automata or non-deterministic finite automata it is decidable in the polynomial time Turing machine.

• A language can be said *PSPACE* complete if that particular language belongs to *PSPACE*.

• For each and every language *PSPACE* hardness is satisfied.

Comment

**Step 2** of 3

Here, user needs to prove that $A_{LBA} \in NL$, but as it is being given that $NP \in NPSPACE$ is *PSPACE*- complete.

With the help of space hierarchy theorem it can be proved that $A_{LBA} \in NL$.

It is being known by the Corollary of the Savitch's theorem that *PSPACE*=*NSPACE*, it implies deterministic space complexity and non-deterministic polynomial space complexity are basically the same.

Space complexity of *PSPACE* is $O(\log n)$ whereas space complexity of the NSPACE is $O(n^k)$ for each and every value of $k_i$.

So, this implies that language is basically accepted by *NSPACE* but it is not accepted by *NL*.

But, it is being known by the space hierarchy theorem that the language is accepted by PSPACE and also language is accepted by NL.

This implies that $NL \in PSPACE$

As, it is being given that $A_{LBA} \in PSPACE$

Hence, it is proved that $A_{LBA} \in NL$

Comment

**Step 3** of 3

Here, user needs to prove that $A_{LBA} \in P$, but as it is being given that $NL \in PSPACE$ is *PSPACE*- complete.

With the help of space hierarchy theorem it can be proved that $A_{LBA} \in P$.

It is being known by the Corollary of the Savitch's theorem that *PSPACE*=*NSPACE*, it implies deterministic space complexity and non-deterministic polynomial space complexity are basically the same.

Poly-time Turing machine is not able to consume space greater than poly-space.

It implies $P \in PSPACE$ and $NP \in NPSPACE$.

As, it is being given that $A_{LBA} \in PSPACE$

Hence, it is proved that $A_{LBA} \in P$

Comment

## Problem

Show that the language *MAX-CLIQUE* from Problem 7.48 is in $P^{SAT}$.

## Step-by-step solution

**Step 1** of 1

- In a graph $G$ clique are maximum of the size $k$ iff it has $k$ clique size and does not have $k+1$ clique size.
- The language $CLIQUE$ is in $NP$, and the language $CLIQUE$ is in $co-NP$. Since $NP \in P^{SAT}$ and $CoNP \in P^{SAT}$, $MAXCLIQUE$ is in $P^{SAT}$ .
- Since $P^{SAT} \in NP^{SAT}$, $MAXCLIQUE$ is in $NP^{SAT}$.

Comment

## Problem

Describe the error in the following fallacious "proof" that P ≠ NP. Assume that P = NP and obtain a contradiction. If P = NP, then *SAT* ? P and so for some k, *SAT* ? TIME($n^k$). Because every language in NP is polynomial time reducible to *SAT*, you have

$$\mathrm{NP} \subseteq \mathrm{TIME}(n^k). \text{ Therefore, } \mathrm{P} \subseteq \mathrm{TIME}(n^k).$$

## Step-by-step solution

**Step 1** of 2

**Consider the following deceptive "proof "that** $\mathrm{P} \neq \mathrm{NP}$.

1. Suppose that $\mathrm{P}=\mathrm{NP}$ and obtain a contradiction.

2. If $\mathrm{P}=\mathrm{NP}$ has given, then for some $\mathrm{k} \in \mathrm{N}$ and $\mathrm{SAT} \in \mathrm{P}$, $\mathbf{SAT} \in \mathbf{TIME}(\mathbf{n}^{\mathbf{k}})$

3. As every language in $\mathrm{NP}$ is polynomial time reducible to $\mathrm{SAT}$ ,then $\mathbf{NP} \subseteq \mathbf{TIME}(\mathbf{n}^{\mathbf{k}})$

4. Due to the assumption $\mathrm{P}=\mathrm{NP}$ which is taken above, $\mathbf{P} \subseteq \mathbf{TIME}(\mathbf{n}^{\mathbf{k}})$.

5. Then $\mathbf{TIME}(\mathbf{n}^{\mathbf{k}}) \subsetneq \mathbf{TIME}(\mathbf{n}^{\mathbf{k+1}})$ is a contradiction with $\mathbf{P} \subseteq \mathbf{TIME}(\mathbf{n}^{\mathbf{k}})$.

Comment

**Step 2** of 2

**The above steps** are used to proof of $\mathrm{P} \neq \mathrm{NP}$, but it consists some error. Now, consider the **step 3**, **here the time needed for calculating the reduction is not taken into account. So, to make this proof error free, the time needed for calculating the reduction must be taken in to account**.

Comments (1)

## Problem

Consider the function $pad: \Sigma^* \times \mathcal{N} \longrightarrow \Sigma^* \#^*$ define the language pad(A, f) as

$$pad(A, f) = \{pad(s, f(m)) | \text{ where } s \in A \text{ and } m \text{ is the length of } s\}.$$

Prove that if $A \in \text{TIME}(n^6)$, then $pad(A, n^2) \in \text{TIME}(n^3)$.

## Step-by-step solution

**Step 1** of 1

For any function $f: N \to N$ and language, $pad(A, f)$ is defined as:

$pad(\text{A}, f) = \{pad(s, f(m)) | \text{ where } s \in \text{A and m is the length of } s\}$.

If $\text{A} \in \text{TIME}(n^6)$ is given then, it is supposed that M be a machine that decide A in time $n^6$.

• Now a machine M' can be considered for $pad(A, n^2)$ that on input x, check if x is of the format $pad(w, |w|^2)$ for some string $w \in \sum^*$. Input x will be rejected if it will not matched. Otherwise, simulate M on $w$.

• The running time of machine M' is $O(|x|^3) + O(|w|^6) = O(|x|^3)$.

Hence, it can be said that $pad(\text{A}, n^2) \in \text{TIME}(n^3)$.

Comment

## Problem

Prove that if NEXPTIME ≠ EXPTIME, then P ≠ NP. You may find the function pad, defined in Problem 9.13, to be helpful.

## Step-by-step solution

**Step 1** of 1

If $\mathbf{EXPTIME} \neq \mathbf{NEXPTIME}$ then $\mathbf{P} \neq \mathbf{NP}$ can be proved by taking its contra positive . If $\mathbf{P} = \mathbf{NP}$ is assumed then $\mathbf{EXPTIME} = \mathbf{NEXPTIME}$ will have to proof.

• Suppose $L \in \mathrm{NTIME}\left(2^{2^c}\right)$ then the following language $L_{\text{pad}} = \left\{ <x, 1^{2|x|^c} >: x \in L \right\}$ is in $\mathrm{NP}$ (in fact in $\mathrm{NTIME}(n)$). This process of adding a string of symbols to each string in the language is called **padding**.

• Hence, if $\mathbf{P} = \mathbf{NP}$ then $L_{\text{pad}}$ is in $\mathrm{P}$ but if $L_{\text{pad}}$ is in $\mathrm{P}$ then $L$ is in $\mathbf{EXPTIME}$.

• To conclude whether an input $x$ is in $L$, it just pads the input and decides whether it is in $L_{\text{pad}}$. This can be achieved by using the polynomial-time machine for $L_{\text{pad}}$.

Therefore, it can be said that if $\mathbf{P} = \mathbf{NP}$ is assumed then $\mathbf{EXPTIME} = \mathbf{NEXPTIME}$ . Thus, from the above explanation it may also be concluded that if $\mathbf{EXPTIME} \neq \mathbf{NEXPTIME}$ then $\mathbf{P} \neq \mathbf{NP}$ .

Comment

## Problem

Define pad as in Problem 9.13.

**a.** Prove that for every A and natural number k, A ? P iff pad(A, $n^k$) ? P.

**b.** Prove that P ≠ SPACE(n).

## Step-by-step solution

**Step 1** of 2

For any language A and function $f : N \to N$ , the language $pad(A, f)$ is defined as:

$(\mathrm{A}, f) = \{pad(s, f(m)) \mid \text{where } s \in \mathrm{A} \text{ and m is the length of } s\}$.

(a) Suppose A be any language and $k \in N$. If $\mathrm{A} \in P$, then $pad(\mathrm{A}, n^k) \in P$ because it can be determined whether $w \in pad(\mathrm{A}, n^k)$ by writing $w$ as $s\#$ where $s$ does not contain the # symbol then it is tested whether $|w| = |s|^k$; and finally it is tested that whether $s \in A$.the implementation of the first test in polynomial time is straight forward. The second test runs in time $\mathrm{poly}(|s|)$ because $|s| \le |w|$, the test runs in time $\mathrm{poly}(|w|)$ and hence is in polynomial in time. If $pad(A, n^k) \in P$ , then $A \in P$ because it can be determined whether $w \in A$ by padding $w$ with # symbols until it has length $|w|^k$ and then test, whether the result is in $pad(A, n^k)$. The above explanation shows that $A \in P$ holds if and only if $pad(A, n^k) \in P$.

Comment

**Step 2** of 2

(b) Assume that $\mathrm{P{=}SPACE}(n)$. Suppose $A$ be a language in $\mathrm{SPACE}(n^2)$ but it will not exist in $\mathrm{SPACE}(n)$ as the space hierarchy theorem says. The language $pad(A, n^2) \in \mathrm{SPACE}(n)$ because it has enough space to run the $O(n^2)$ space algorithm for $A$, using that is linier in the padded language. Because of the assumption, $pad(A, n^2) \in P$ , hence $A \in \mathrm{P}$ as discussed above. Hence $\mathrm{A} \in \mathrm{SPACE}(n)$ by taking assumption once again. But that is a contradiction. Hence, It can be said that $\mathrm{P} \ne \mathrm{SPACE}(n)$.

Comment

## Problem

Prove that $TQBF \notin \mathrm{SPACE}(n^{1/3})$.

## Step-by-step solution

**Step 1** of 1

The **space hierarchy theorem** says "if g is a space-constructible ( $1^n \to 1^{g(n)}$ can be computed in space $O(g(n))$, $f(n) = o(g(n))$, then $\mathbf{SPACE}(f(n)) \subsetneq \mathbf{SPACE}(g(n))$". So, from the space hierarchy theorem it can be said that there exists a language $L$, which is **solvable in linear space but it cannot be solved by sub-linear space**.

• Since, $\mathbf{TQBF}$ is space complete then $L$ can be reduced to $\mathbf{TQBF}$ in log space. Therefore, if $\mathbf{TQBF} \in \mathbf{SPACE}\left(\mathbf{n}^{1/3}\right)$, then $L \in \mathbf{SPACE}\left(\left(\mathbf{n}^{c}\right)^{1/3} + \log(\mathbf{n})\right)$.

• Now suppose, if $0.33 < 1/c$, **there exists a contradiction**.

**Thus, from the above result it can be said that** $TQBF \notin SPACE\left(n^{1/3}\right)$.

Comment

## Problem

Read the definition of a 2DFA (two-headed finite automaton) given in Problem 5.26. Prove that P contains a language that is not recognizable by a 2DFA.

## Step-by-step solution

**Step 1** of 2

**Suppose** $L = \{p \mid \text{either } p = 0x \text{ for some } x \in B_{TM}, or\ p = 1y \text{ for some } y \notin B_{TM}\}$.

• It can be designed a Turing machine $S$ and $S$ may be defined as:

$S$ : On input, write 0 followed by $\langle M, p \rangle$ in the tapes and halts. Then it is easy to check that:

$\langle M, p \rangle \in B_{TM} \Leftrightarrow output\ of\ Q \in L$

**Thus**, a mapping reduction of $B_{TM}$ to $L$ or $B_{TM} \leq_m L$ can be obtained .

• Now, a Turing machine(TM) $R$ can be formed, which shows the functionality $B_{TM} \leq_m \overline{L}$. The Turing machine(TM) $R$ can be defined as:

$R$ : On input, write 1 followed by $\langle M, p \rangle$ in the tapes and halts. Then it is easy to check that:

$\langle M, p \rangle \in \overline{B}_{TM} \Leftrightarrow output\ of\ R \in L$

• Similarly,

$\langle M, p \rangle \in B_{TM} \Leftrightarrow output\ of\ R \in \overline{L}$

**Thus**, a mapping reduction of $B_{TM}$ to $\overline{L}$ can be obtained .

Comment

**Step 2** of 2

**Since**, $B_{TM} \leq_m L$ and $\overline{B}_{TM} \leq_m \overline{L}$. This show that $\overline{L}$ is non Turing recognizable because $B_{TM}$ is **non Turing recognizable**. Similarly, since $B_{TM} \leq_m \overline{L}$ and $\overline{B}_{TM} \leq_m L$. So, this allows that $L$ is non Turing recognizable. **Therefore, the above explanation shows that " $P$ contains a language which is not recognizable by a 2DFA"**.

Comment

## Problem

Let $E_{\mathsf{REX}\uparrow} = \{\langle R \rangle |$

## Step-by-step solution

**Step 1** of 1

As it is known that **"the problem of any $\mathbf{EXSPACE}$ - complete cannot be in $\mathbf{PSPACE}$ or** In other words, it can be said that " $\mathbf{PSPACE} \subsetneq \mathbf{EXSPACE}$ ".

• Now, consider the statement which is given below:

$E_{REX\uparrow} = \{\langle R \rangle \mid R \textbf{ is a regular expression with exponentiation and } L(R) = \varnothing\}$

• The above statement show that $R$ **is a regular expression** and the language, which contains this regular expression, consists NULL value.

• By applying the concept of intractability, it can be said that $E_{REX\uparrow}$ **is intractable** because it can be demonstrated in such a manner that it is complete for the class $\mathbf{EXSPACE}$.

• Now, from the above discussion, it may be concluded that "any $\mathbf{EXSPACE}$ **- complete problem cannot be in** $\mathbf{PSPACE}$ **and it is much less in** $P$".

• Otherwise, $\mathbf{PSPACE}$ will be same as $\mathbf{EXSPACE}$, which is contradicting the corollary that is discussed above.

As it is explained above that" any $\mathbf{EXSPACE}$ **- complete problem is much less in** $P$". Therefore, it can be said that $\mathbf{E}_{\mathbf{REX}\uparrow} \in \mathbf{P}$.

Comment

## Problem

Define the ***unique-sat*** problem to be

*USAT* = { $\langle \phi \rangle \mid \phi$ is a Boolean formula that has a single satisfying assignment}.

Show that *USAT* ? $P^{SAT}$.

## Step-by-step solution

**Step 1** of 1

**Given:**

$USAT = \{\langle \phi \rangle \mid \phi\}$, is basically a Boolean formula which is use for satisfying the single assignment.

**Proof:**

Here, it is to be proved that every Boolean formula which is satisfied has minimum one assignment which is bounded.

For proving this user need to satisfy the assignment of the *USAT* formula which is defined syntactically which the help of propositional logic.

$USAT \in \mathrm{P}^{SAT}$

**Construction:**

Here, user is proving that *USAT* should have only single truth value assignment which can be proved with the help propositional logic.

• User need to define a class of bonded truth value.

• It is to be proved that formula $\phi$ should have minimum one assignment which is bounded. For proving this there is one assignment which is chosen that is $l_\phi$ which should be syntactically defined.

• Therefore, characterization of the assignment which is uniquely satisfied is $\phi \in \mathrm{USAT}$ and it is proved by the definition which is $l_\phi$.

• $l_\phi$, constraint is use for satisfying all the assignment of $\phi$ this is because it contains information for the truth value assignment which is use for satisfying the value of $\phi$.

• The assignment $l_\phi$ is syntactically defined with the help of propositional logic for each and every value of the variable *p*, substitution of the value p is the variable free formula.

• Here, $l_\phi$ is satisfying all the truth value assignment $\phi$.

• $l_\phi$, contains information for the truth value assignment which is use for satisfying the value of $\phi$ and even it help in determining the other truth value which is use for falsifying the value of $\phi$.

It is proved that unique assignments in USAT are basically bounded and can be syntactically defined with the help of propositional logic.

Hence, it is proved that $USAT \in \mathrm{P}^{SAT}$

Comment

## Problem

Prove that an oracle C exists for which $NP^C \neq coNP^C$

## Step-by-step solution

**Step 1** of 2

It is known that an oracle $A$ exists such that $P^A \neq NP^A$ and $L_A \notin P^A$. For a given oracle $A$, $L_A$ may be defined as:

$$L_A = \{w : |w| = |x| \text{ for some } x \in A\}$$

Then, it is clear that $L_A$ is in $NP^A$.So, finally $L_A \notin coNP^A$ will have to proof (that is, same as $\overline{L}_A \notin NP^A$).

• For this purpose, first of all $A$ will be constructed and $M_1, M_2, \cdots$ is now a list of all nondeterministic polytime oracle TMs, instead of all deterministic polytime oracle TMs.

Comment

**Step 2** of 2

**For Stage** $i$ **,** choose $n$ and run $M_i$ in input $1^n$. It is respond NO to the query if $M_i$ queries a string $y$ whose status has not yet been determined.

• If there does not exist any computation and also, if $M_i$ does not accept $1^n$ under these conditions then $M_i$ is forced to make a mistake. It is performed because $A$ can be maintained permanently that contains no string of length $n$. Then, $1^n \in \overline{L}_A$, but $M_i$ does not accept $1^n$.

If, on the other hand, there is an accepting computation for $M_i$ with input $1^n$, then it may be noted that this accepting computation can only query polynomially many strings $y$.

• Hence, there is a string $x$ of length $n$ which this computation does not query. We specify that this string $x$ is in $A$ and no other string of length $n$ is in $A$.

• $M_i$ Still has the same accepting computation on input $1^n \notin \overline{L}_A$. So makes a mistake in this case also. So, it may be concluded from the above explanation is that, an oracle $C$ exists for which $NP^c \neq coNP^c$.

Comment

## Problem

A k-***query oracle Turing machine*** is an oracle Turing machine that is permitted to make at most k queries on each input. A k-query oracle Turing machine M with an oracle for A is written $M^{A,k}$. Define $P^{A,k}$ to be the collection of languages that are decidable by polynomial time k-query oracle Turing machines with an oracle for A.

**a.** Show that $\mathrm{NP} \cup \mathrm{coNP} \subseteq \mathrm{P}^{SAT,1}$.

**b.** Assume that $\mathrm{NP} \neq \mathrm{coNP}$. Show that $\mathrm{NP} \cup \mathrm{coNP} \subsetneq \mathrm{P}^{SAT,1}$.

## Step-by-step solution

**Step 1** of 2

In this user need to prove that union of the *NP* and *coNP* can be decided in polynomial time by using the oracle of the *SAT* problem.

This implies $\mathrm{NP} \cup \mathrm{coNP} \subseteq \mathrm{P}^{SAT,1}$

It is being known that the oracle problem *SAT* is basically the *NP*-complete problem. *NP* language is basically encoded in the polynomial time *SAT*.

As, it is known that $\mathrm{L} \subseteq \mathrm{NP}$ and $\mathrm{P}^{A,K} \in \mathrm{NP}$.

This implies *L* and $\mathrm{P}^{A,K} \subseteq \mathrm{NP}$ it can reduced in Poly-time by the oracle problem *SAT*.

So, $\mathrm{NP} \subseteq \mathrm{P}^{SAT}$

Similarly,

As, it is known that $\mathrm{L} \subseteq \mathrm{NP}$ and $\mathrm{P}^{A,K} \in \mathrm{NP}$.

This implies *L* and $\mathrm{P}^{A,K} \subseteq \mathrm{NP}$ it can reduced in Poly-time by the oracle problem *SAT*.

Even, $\neg \mathrm{L}\left(\in \mathrm{NP} \subseteq \mathrm{P}^{SAT}\right)$ can be reduced in Poly-time by the oracle problem *SAT*.

So, $\mathrm{coNP} \subseteq \mathrm{P}^{SAT}$

In case of *NP* "yes" answer is checked by the oracle Turing machine and in case of *coNP* "no" answer is check by the oracle Turing machine in polynomial time.

It is being known that for each and every *NP* complete problem there is *coNP* complete problem.

Suppose, *coNP* and *NP* are equal then in that case the polynomial collapsed to either *NP* or *coNP*. But as it is shown earlier that $\mathrm{NP} \subseteq \mathrm{P}^{SAT}$ and $\mathrm{coNP} \subseteq \mathrm{P}^{SAT}$.

Here union operation is done between *NP* and *coNP* as here, the output is in either "yes" or "no".

So, union is computed in the polynomial time.

Hence, $\mathrm{NP} \cup \mathrm{coNP} \subseteq \mathrm{P}^{SAT,1}$

Comment

**Step 2** of 2

In this user need to prove that union of the *NP* and *coNP* can be decided in polynomial time by using the oracle of the *SAT* problem.

This implies $\mathrm{NP} \cup \mathrm{coNP} \not\subset \mathrm{P}^{SAT,1}$

It is being known that the oracle problem *SAT* is basically the *NP*-complete problem. *NP* language is basically encoded in the polynomial time *SAT*.

As, it is known that $\mathrm{L} \subseteq \mathrm{NP}$ and $\mathrm{P}^{A,K} \in \mathrm{NP}$.

This implies *L* and $\mathrm{P}^{A,K} \subseteq \mathrm{NP}$ it can reduced in Poly-time by the oracle problem *SAT*.

So, $\mathrm{NP} \subseteq \mathrm{P}^{SAT}$

Similarly,

As, it is known that $\mathrm{L} \subseteq \mathrm{NP}$ and $\mathrm{P}^{A,K} \in \mathrm{NP}$.

This implies *L* and $P^{A,K} \subseteq NP$ it can reduced in Poly-time by the oracle problem *SAT*.

Even, $\neg L\left(\in NP \subseteq P^{SAT}\right)$ can be reduced in Poly-time by the oracle problem *SAT*.

So, $coNP \subseteq P^{SAT}$

In case of *NP* "yes" answer is checked by the oracle Turing machine and in case of *coNP* "no" answer is check by the oracle Turing machine in polynomial time.

It is being known that for each and every *NP* complete problem there is *coNP* complete problem.

In this *coNP* and *NP* are not equal.

$NP \subseteq P^{SAT}$ and $coNP \subseteq P^{SAT}$.

So, union is not computed in the polynomial time.

Hence, $NP \cup coNP \not\subset P^{SAT,1}$

Comment

## Problem

Suppose that A and B are two oracles. One of them is an oracle for *TQBF*, but you don't know which. Give an algorithm that has access to both A and B, and that is guaranteed to solve *TQBF* in polynomial time.

## Step-by-step solution

**Step 1** of 3

An Oracle Turing machine is a type of Turing machine having many different tapes and these tapes are called oracle tapes. The different states may be represented by $q_{states}$.

A TQBF is True Qualified Boolean formula. To Show TQBF in polynomial problem which has access to turing machines A and B can be solved by using Baker Gill Solovay Theorem. The problem can be broken up in two parts of algorithm:

Comment

**Step 2** of 3

**For Oracle A:**

1. Consider TQBF has access to A then A=TQBF.

2. In the given problem it can be proved by showing $P^A = NP^B$.

3. As $TQBF$ is $PSPACE$ problem,

4. Hence $PSPACE \subseteq P^{TQBF} \, and \, PSPACE^{TQBF} \subseteq PSPACE$

5. Therefore, combining both result $P^A = NP^B$.

Comments (1)

**Step 3** of 3

**For Oracle B:**

1. For oracle $B$ it is required to show that $P^B \neq NP^B$.

2. For this define a language $L_B$ as: $L_B = \{0^n \mid w \in \{0,1\}\} \; Here, \; B(w) = 1$

3. For any oracle $B$ the defined language $L_B$ is $NP^B$.

4. Now the machine's output will be $1$ if $x = 0^n$ with $|w| = |x|$.

5. Here $TQBF \in NL$ this shows that $PSPACE \in NL$

6. By using hierarchy theorems and Baker Gill which states that $NL \subsetneq PSPACE$.

7. It shows that $P^B \neq NP^B$.

Using both two parts of algorithm result it is sure that $TQBF$ is in polynomial time for both oracle $A$ and $B$.

Comments (1)

## Problem

Recall that you may consider circuits that output strings over {0,1} by designating several output gates.

Let $add_n : \{0,1\}^{2n} \longrightarrow \{0,1\}^{n+1}$ take two n bit binary integers and produce the n+1 bit sum. Show that you can compute the $add_n$ function with O(n) size circuits.

## Step-by-step solution

**Step 1** of 1

Consider the following *add* functions:

$$add_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$$

This function basically takes two *n* bit binary integer and thus it produces *n*+1 bit sum.

User need to compute the complexity of the *add* function.

Speed of the circuits depends upon the gate which has been chosen. If the complexity is more than at that time, speed should be increased.

So, there is the trade-off between the space and time complexity. It implies that if it is taking less time than speed is more or vice versa.

In case of binary adder, input is two bit binary integer. Suppose there are *n* digits then it is represented with the help of *n* line. Summation of the two *n* bit binary number is equal to *n*+1 bit binary number.

Sum generated by the two *n* bit binary integer is *n*+1 bit long which can be represented in the form of $O(n)$.

As $n + 1$, $n + 2 \ldots$ $n + n$ are represented in form of order of *n*.

Hence, it is proved that complexity of the $add_n$ function is $O(n)$.

Comment

## Problem

Define the function $majority_n : \{0,1\}^n \longrightarrow \{0,1\}$ as

$$majority_n(x_1, \ldots, x_n) = \begin{cases} 0 & \sum x_i < n/2\,; \\ 1 & \sum x_i \geq n/2\,. \end{cases}$$

Thus, the $majority_n$ function returns the majority vote of the inputs. Show that $majority_n$ can be computed with:

**a.** $O(n^2)$ size circuits.

**b.** O(n log n) size circuits. (Hint: Recursively divide the number of inputs in half and use the result of Problem 9.23.)

## Step-by-step solution

**Step 1** of 2

a) Let the number of inputs taken is $n$. A **bubble-sort** can be implemented as a circuit. It is used tocompare two bits and after comparing, reordering them if necessary is rather easy. The inputs can be called as $x_1, x_2$ and the outputs can be called as $y_1, y_2$. A sub-circuit can be written which accomplishes this as $y_1 = OR(x_1, x_2)$ and $y_2 = AND(x_1, x_2)$ .**This circuit contains a size of two**.

• Now, the action of the bubble-sort algorithm can be mimicked on an array. It can be implemented one step at position to be the $n$ input, $n$-output sub-circuit that passes through all the inputs taken as $< k\ and \geq k+1$ are unchanged.

• Now, the compare-swap sub-circuit, which is described above, on $< k\ and\ \geq k+1st$ input can be used to generate the $kth$ and $k+1st$ output. This still has size two. Now, **a pass** can be implemented as the serial concatenation of steps for each of $k = 1, 2, \ldots, n-1$, which has a size $(n-1)*2$.

• A bubble-sort can be Proceed to implement as the serial concatenation of $n$ passes. Therefore, this gives a size $\mathbf{n(n-1)*2 = O(n^2)}$.

**Therefore, it can be said that $\mathbf{majority_n}$ can be computed in $O(n^2)$ size circuits.**

Comment

**Step 2** of 2

b) Let the **number of inputs taken** is $n$. A **Merge-sort** can be implemented as a circuit. It is used to compare two bits after recursively dividing the given inputs in to half. The total time taken here (to divide the inputs into equal halves iteratively)is $\log n$.

• Finally at the last, the inputs can be called as $x_1, x_2$ and the outputs can be called as $y_1$.

• Now,**the action of the merge-sort algorithm** can be mimicked on an array. It can be implemented one step at position to be the $n$ input, $n/2$**-output sub-circuit**.

• Now, **a pass** can be implemented as the serial concatenation of steps, which has a size $n \log n$. Therefore, this gives a size $n * \log n = O(n \log n)$.

**Therefore, it can be said that $\mathbf{majority_n}$ can be computed in $O(n \log n)$ size circuits.**

Comment

## Problem

Define the function $majority_n$ as in Problem 9.24. Show that it may be computed with O(n) size circuits.

Problem 9.24

Define the function $majority_n : \{0,1\}^n \longrightarrow \{0,1\}$ as

$$majority_n(x_1, \ldots, x_n) = \begin{cases} 0 & \sum x_i < n/2\,; \\ 1 & \sum x_i \geq n/2\,. \end{cases}$$

Thus, the $majority_n$ function returns the majority vote of the inputs. Show that $majority_n$ can be computed with:

**a.** $O(n^2)$ size circuits.

**b.** O(n log n) size circuits. (Hint: Recursively divide the number of inputs in half and use the result of Problem 9.23.)

## Step-by-step solution

**Step 1** of 1

Consider the number of inputs taken is $n$. A **bubble-sort** can be implemented as a circuit. It is used to compare two bits and after comparing, reordering them if necessary is rather easy. The inputs can be called as $x_1, x_2$ and the outputs can be called as $y_1, y_2$. A sub-circuit can be written which accomplishes this as $y_1 = OR(x_1, x_2)$ and $y_2 = AND(x_1, x_2)$ . **This circuit contains a size of two**.

• Now, the action of the bubble-sort algorithm can be mimicked on an array. It can be implemented one step at position to be the $n$ input, $n$-output sub-circuit that passes through all the inputs taken as $< k\ and \geq k+1$ are unchanged.

• Now, the compare-swap sub-circuit, which is described above, on $< k\ and\ \geq k+1st$ input can be used to generate the $kth$ and $k+1st$ output. This still has size two. Now, **a pass** can be implemented as the serial concatenation of steps for each of $k = 1, 2, \ldots, n-1$, which has a size $(n-1)*2$.

• A bubble-sort can be Proceed to implement as the serial concatenation of one passes. Therefore, this gives a size $\mathbf{1(n-1)*2 = O(n)}$.

**This** $O(n)$ complexity can be achieved only when the **input taken is already in sorted order**. In other word it can be said that, if it shows **a best case behavior**. **Therefore, it can be said that** $\mathbf{majority_n}$ **can be computed in** $O(n)$ **size circuits.**

Comment

## Problem

Show that a circuit family with depth O(log n) is also a polynomial size circuit family.

## Step-by-step solution

**Step 1** of 2

A circuit family of depth $O(\log n)$ can be obtained by taking an equivalent polynomial size family of formulas. **"To convert a formula $\mu$ with $h$ leaves to a similar circuit of depth $O(\log h)$," is sufficient to show the above statement.** Here, it may be assumed that fan-in value of all the nodes is 2 and not gates are pushed to the leaves.

• Now, the proof of $h \geq 4$ (can be done using the induction hypothesis) that is a formula $\mu$ with $h$ leaves is similar to a formula $\mu'$ with a maximum depth of $C \log_2 h$ .where the value of the constant C will be further determined.

• If $h \leq 4$, suppose $\mu' = \mu$. Otherwise apply the concept of tree which says that every tree with $m \geq 2$ leaves has a sub-tree with between $m/3$ and $2m/3$. By applying this concept, the tree structure of $\mu$ acquire a sub-formula $\beta$ with between $h/3$ and $2h/3$.

Comment

**Step 2** of 2

**Suppose** $\hat{\mu}(y)$ be $\mu$ with the sub-formula $\mu$ is replace by a new variable $y$. Thus, $\mu$ is similar to $\hat{\mu}(\beta)$ and $\mu$ is equivalent to $\mu_1$ that is given by:

$$\mu_1 = \left(\beta \wedge \hat{\mu}(1)\right) \vee \left(\neg\beta \wedge \hat{\mu}(0)\right)$$

Here, $\hat{\mu}(1)$ and $\hat{\mu}(0)$ each contains maximum $2h/3$ leaves which is variable.

• Finally, suppose $\mu$ and $\mu'$ with the equivalent of the sub formulas $\beta$, $\hat{\mu}(1)$ and $\hat{\mu}(0)$ interchanged by similar small depth formula given by the induction hypothesis.

• **Thus, the depth of $\mu'$ is maximum** $C \log_2\left((2/3)h\right) + 3$. This is maximum of $C \log_2 h$ provided $C \geq 3 \big/ \log_2\left(\frac{3}{2}\right)$.

• Thus, from the above explanation it can be said that "**A circuit family of depth $O(\log n)$ can be obtained by taking an equivalent polynomial size family of formulas**".

Comment

## Problem

Show that 12 is not pseudoprime because it fails some Fermat test.

## Step-by-step solution

**Step 1** of 2

A number is said to be pseudoprime if the number passes Fermat test. This means the number should be prime related to all the numbers that are less than the given number.

Fermat primality test is a test used for checking whether the number is prime or not. A number is said to be prime if the number satisfies following condition:

$$p^{n-1} \equiv 1 \pmod{n}$$

where $p \geq 1$ and $n > p$

The equation given above can be written as:

$$p^{n-1} - 1 = nk$$

where $k$ is an positive integer.

Comment

**Step 2** of 2

Fermat test for 12 is given below:

$$4^{12-1} - 1 = 12k$$
$$4^{11} - 1 = 12k$$

$$4194304 - 1 = 12k$$
$$4194303 = 12k$$

When 4194303 is divided with 12k, this will provide remainder 3. So, 12 fails Fermat test. So, 12 is not pseudoprime.

Comment

## Problem

Prove that if A $\leq_L$ B and B is in NC, then A is in NC.

## Step-by-step solution

**Step 1** of 2

**If** $A \leq_L B$ **and** $B$ **is in** $NC$ **then it can be proved that** $A$ **is also in** $NC$. This can be achieved by using the fact of circuit evaluation. In other word, this can be achieved by showing that" **the problem of circuit evaluation is** $P$ **complete".**

For a circuit $C$ and input string $w$, the value of $C$ on $w$ can be written as $C(w)$. **Suppose**

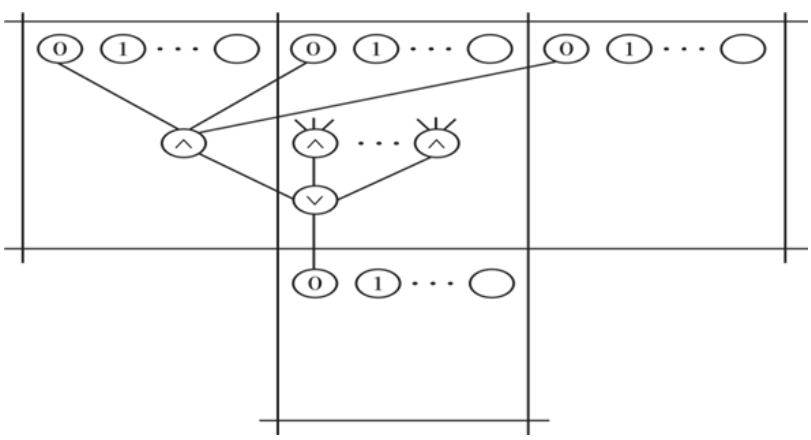
$$CIRCUIT-VALUE = \{\langle C, x\rangle \mid C \text{ is a Boolean circuit and } C(x) = 1\}$$

Comment

**Step 2** of 2

Consider the given theorem, which says that "**suppose** $t: M \to M$ **be a function, where** $t(m) \geq m$**. If** $W \in TIME(t(m))$**, then the complexity of the circuit** $A$ **is given by** $O(t^2(m))$

• Now, consider the figure which is given below:




• The above figure shows the construction of the theorem, which is discussed above. It shows the way of **reduction of a language** $W$ **(which is in** $P$**)** to $CIRCUIT-VALUE$.

• On input $w$, the production of a circuit takes place by the reduction. The process reduction simulates the Turing machine for $W$ in polynomial time. The $w$ itself can be taken as an input to the circuit.

• A $\log$ space is used to carried out the reduction because **the circuit produced by it contains a repetitive and a simple structure.**

Hence, **it shows that "** $CIRCUIT-VALUE$ **is** $P$**-complete" and the circuit produced by it has a repetitive structure.** Therefore it can be said that "**If** $A \leq_L B$ **and** $B$ **is in** $NC$ **then** $A$ **is also in** $NC$."

Comment

## Problem

Show that the parity function with n inputs can be computed by a branching program that has O(n) nodes.

## Step-by-step solution

**Step 1** of 1

**A branching program** is defined as "**a directed acyclic graph** where labels of all the nodes are maintained by the variables, except for two output nodes labeled 1 or 0. Here, all the nodes **whose labels are maintained by the variables** are called **query nodes**. Every query nodes consists of two outgoing edges: one is labeled 1 and another one is labeled 0.

• So, from the definition of branching program as defined above "the n-input parity function can be computed by a branching program that consists $O(n)$ nodes.

• It can be achieved by **building a binary tree of gates that compute XOR function**, **where XOR function is used as equivalent to the parity function. The implementation of each XOR gate can be done by using two AND's, two NOT's and one OR gates.**

• As the implementation of XOR function consists a binary tree of different gates. Here, the output node of binary tree is labeled 1 and 0, which acts a branching program.

**Therefore**, from the above explanation it can be said that "**the n-input parity function can be computed by a branching program that consists** $O(n)$ **nodes**".

Comment

## Problem

Show that the majority function with n inputs can be computed by a branching program that has O($n^2$) nodes.

## Step-by-step solution

**Step 1** of 2

**A branching program**is defined as "**a directed acyclic graph** where the variables are used to label all the nodes except only two output nodes which is labeled $0$ and $1$". The **query nodes** are defined as all the nodes whish are labeled by the variables. All the query nodes consists two outgoing edges, labeled as 0 and 1. Both output nodes doesn't consists outgoing edges.

Now, consider about the **function majority** $\left(majority_n : \{0,1\}^n \to \{0,1\}\right)$, that is defined as:

$$\begin{aligned} majority_n\left(x_1, x_2, ..., x_n\right) &= 0 \ if \sum x_i < n/2; \\ &= 1 \ if \sum x_i \geq n/2. \end{aligned}$$

The computation of the **majority function** can be done by using **a branching program**, which consist $O\left(n^2\right)$nodes.

Comment

**Step 2** of 2

Now, suppose the number of inputs taken is $n$. **A bubble-sort can be implemented as a circuit**. It is used to compare two bits and after comparing, reordering them if necessary is rather easy. The inputs can be called as $x_1, x_2$ and the outputs can be called as $y_1, y_2$. A sub-circuit can be written which accomplishes this as $\mathbf{y_1 = OR\left(x_1, x_2\right)}$and $\mathbf{y_2 = AND\left(x_1, x_2\right)}$ . This will be act as a part of **branching program**. This circuit contains a size of two.

• Now, the action of the bubble-sort algorithm can be mimicked on an array. It can be implemented one step at position to be the $n$input, $n$-output sub-circuit that passes through all the inputs taken as $< k \ and \geq k+1$ are unchanged.

• Now, the compare-swap sub-circuit, which is described above, on $< k \ and \ \geq k+1st$ input can be used to generate $kth$ and $k+1st$ output. This still has size two. Now, **a pass** can be implemented as the serial concatenation of steps for each of $k = 1, 2, ..., n-1$, which has a size $(n-1)*2$.

• A bubble-sort can be Proceed to implement as the serial concatenation of $n$passes. Therefore, this gives a size $\mathbf{n(n-1)*2 = O\left(n^2\right)}$.

**Here, AND gates and OR gates are used to construct the branching program. Therefore, it can be said that "a branching program with $O\left(n^2\right)$ nodes can be used to compute the majority function with $n$ inputs.**

Comment

## Problem

Show that any function with n inputs can be computed by a branching program that has $O(2^n)$ nodes.

## Step-by-step solution

**Step 1** of 4

**A program** is defined as a branching program if "**a directed graph, which also shows an acyclic** property, where labels of all the given nodes are maintained by the variables. These variables will not be used for two output nodes which are labeled 1 or 0. Here, all the nodes **whose labels are maintained by the variables** are also known as **query nodes**. Every query nodes consists of two edges which is outgoing from itself. In which one output node is labeled as 1 and another one is labeled as 0.

Comment

**Step 2** of 4

Now, consider the **majority function** $\left(majority_n : \{0,1\}^n \to \{0,1\}\right)$, which is defined as:

$$\begin{aligned} majority_n\left(x_1, x_2, \ldots, x_n\right) &= 0 \ if \sum x_i < n/2; \\ &= 1 \ if \sum x_i \geq n/2. \end{aligned}$$

As it is known "the computation of the **majority function** can be done by using **a branching program** of $O\left(2^n\right)$nodes"

Comment

**Step 3** of 4

Again, consider the function **parity**; it is known that "**a branching program with** $O(n)$ **gates may be used to compute the n-input parity function".**

• It can be achieved by **building a binary tree of gates which is used to calculate function XOR**, **where the function XOR is used as equivalent to the parity function. The implementation of each XOR gate can be done by using two AND's, two NOT's and one OR gates.**

• It is known that each **AND's, OR gates takes two nodes as an input and produces a single output w**hich again considered as an input of another gate because a binary tree of gates is considered**.**

• Finally, the total number of nodes, which are used in computation, is an order of $2^n$ or $O\left(2^n\right)$.

Comment

**Step 4** of 4

As discussed above, the **majority** and the **function parity, which** takes $n$-inputs, can be obtained by a branching program, which consist $O\left(2^n\right)$ nodes. Therefore, **as the definition of branching program says (which is explained above), it can be said that any function, that consists** $n$**-inputs, can be computed by using a branching program which consist** $O\left(2^n\right)$ **nodes.**

Comments (1)

## Problem

Show that BPP $\subseteq$ PSPACE.

## Step-by-step solution

**Step 1** of 1

Consider $M$ be a probabilistic $TM$ which runs on polynomial time.

Therefore, $M \in BPP$. $M$ can be modified so that $M$ makes exactly $n^k$ coin tosses in each branch of its computation, for any constant $k$.There are total of $2^{(n^k)}$ computation paths. Hence, problem which is determining probability that $M$ accepts its input is reduces to counting how many branches, $B$, are accepting and comparing this number with $P = \left(\frac{3}{4}\right).2^{(n^k)}$.

If $B \geq P$, then accept; otherwise reject. Now the given deterministic task can be performed in the polynomial space by generating all possible paths sequentially following *M*'s program but recycling the space used by the previous path.

Hence, the problem of $BPP$ is converted into $PSPACE$. Therefore, $BPP \subseteq PSPACE$.

Comment

## Problem

Let A be a regular language over {0,1}. Show that A has size–depth complexity (O(n),O(log n)).

## Step-by-step solution

**Step 1** of 1

Suppose a **language A** is defined in $\{0,1\}$ that consist every strings with an odd number of one's. The parity function computation can be used to test membership in A. The standard AND, OR and NOT operations can be used **to implement the two parity gate** $a \oplus b$ as $(a \wedge \neg b) \vee (\neg a \wedge b)$.

• Suppose $a_1, a_2, \ldots, a_n$ be taken as the input to the circuit. There are many ways define a circuit with $O(n)$ size. One way to get a circuit for parity function is to construct gates $g_i$ whereby $g_1 = a_1$ and $g_i = a_i \oplus g_{i-1}$ for $i \le n$. This construction uses $O(n)$ size and depth.

• Another way to do this, by **building a binary tree of gates** that computes the XOR function, where the XOR function is the same as the parity function and then implements **each XOR gate with two NOTs, two ANDs and one OR gates**. This construction uses $O(n)$ and $O(\log n)$ depth.

This construction is a significant improvement because it **uses exponentially less parallel time** than does the preceding construction. **Thus, the size-depth complexity of A is** $(O(n), O(\log n))$.

Comment

## Problem

A ***Boolean formula*** is a Boolean circuit wherein every gate has only one output wire. The same input variable may appear in multiple places of a Boolean formula. Prove that a language has a polynomial size family of formulas iff it is in $NC^1$. Ignore uniformity considerations.

## Step-by-step solution

**Step 1** of 2

**A Boolean formula** is defined as a Boolean circuit which consist only a single output wire for every input gate. The Boolean formula may consists the same input variable at many places. Here, it can be shown that **a polynomial size family of formulas can be used to compute all the languages in** $\mathbf{NC^1}$.

A normal induction hypothesis on $d$ is used to show that "a formula, whose size is less than $O(2^d)$ is similar to every Boolean circuit of depth $d$". For each step of the induction, the circuit's output gate is considered in such a way that the maximum fan-in value acquired is 2. The induction hypothesis can also be applied to each input gate.

• The $n$th circuit $C_n$ has depth $O(\log n)$ in an $\mathbf{NC^1}$ circuit family. Therefore, the equivalent formula has size $2^{O(\log n)} = n^{O(1)}$ that is polynomial in size.

Comment

**Step 2** of 2

**To prove its converse**, first it need to proof that every tree with $h \geq 2$ leaves consist a sub-tree with between $h/3$ and $2h/3$ leaves.

• Suppose a binary tree is denoted by $B$ with $h \geq 2$ leaves. Beginning at the parent (root) of $B$, and traverse towards the child's (leaves), always taking a sub-tree with minimum half of the number of the existing sub-tree.

• Finally stop this iteration when a sub-tree $B'$ is reached which consists at most $2h/3$ leaves. Then, $B'$ will contain minimum of $h/3$ leaves as the previous sub-tree consists more than $2h/3$ leaves. Thus, the desired sub-tree is $B'$.

**Thus**, from the above explanation, it can be said that **a polynomial size family of formulas can be used to compute all the languages in** $\mathbf{NC^1}$.

.

Comment

## Problem

A k-***head pushdown automaton*** (k-PDA) is a deterministic pushdown automaton with k read-only, two-way input heads and a read/write stack. Define the class $PDA_k$ = {A| A is recognized by a k-PDA}. Show that $\mathrm{P} = \bigcup_k \mathrm{PDA}_k$. (Hint:Recall that P equals alternating log space.)

## Step-by-step solution

**Step 1** of 3

A deterministic pushed down automation, which consists $k$ read-only, a read write stack **and two ways input heads**, can be defined as a $k$**-head pushed-down automation** ( $k-\mathrm{PDA}$ ). Consider the **class** $\mathrm{PDA}_k$, which is defined as:

$$\mathrm{PDA}_k = \{A \mid A \, is \, recognized \, by \, a \, k-\mathrm{PDA}\}$$

Now, by using the above given facts, $\mathrm{P}=\bigcup_k \mathrm{PDA}_k$ has to be proved. It can be achieved by using, $ASPACE(s(n)) = TIME\left(2^{O(s(n))}\right)$, which shows that $\mathrm{P}$ is alternating log-space.

Comment

**Step 2** of 3

Now, a machine $M$, with deterministic time $2^{O(s(n))}$, is constructed. It is used to simulate an alternating space $O(s(n))$ machine $S$. If an input $q$ is given to the simulator $M$, a graph is constructed by the simulator for the computation of $S$ on $q$.

• The nodes are configured for $S$ on $q$ which use maximum $ls(n)$ space, where $l$ is defined as the constant factor approximation for $S$.

• Edges are drawn from a configuration to those other configuration which can be generated in a single move of $S$.

• After the construction of graph, $M$ iteratively scans it and marks those configurations which are accepting.

• Initially the acceptance configuration is marked. After that all the universal branching is marked is all of its children marked as an accepting state. Machine $S$ continued marking and scanning until no additional nodes are marked on scan.

Comment

**Step 3** of 3

As it is given that, $s(n) \geq \log n$, the configuration's number of $S$ on $q$ is $2^{O(s(n))}$. Hence the configuration graph's size is given by $2^{O(s(n))}$ and its construction may be done in $2^{O(s(n))}$ time.

• It takes roughly the same time to **scan the graph once**. Here, the **number of scans** is **equal to the maximum number of nodes in the graph**. Hence, the total time used is $2^{O(s(n))}$.

• Now, from the above discussion it can be said that $ASPACE(s(n)) = TIME\left(2^{O(s(n))}\right)$. It also shows, $P$ **is alternating log-space.**

**Hence, for** $\mathrm{PDA}_k = \{A \mid A \, is \, recognized \, by \, a \, k-\mathrm{PDA}\}$, it can be said that $\mathrm{P}=\bigcup_k \mathrm{PDA}_k$.

Comment

## Problem

LetM be a probabilistic polynomial time Turing machine, and let C be a language where for some fixed $0 < \epsilon_1 < \epsilon_2 < 1,$

**a.** $w \notin C$ implies $\Pr[M \text{ accepts } w] \leq \epsilon_1$, and

**b.** $w \in C$ implies $\Pr[M \text{ accepts } w] \geq \epsilon_2$.

Show that $C \in \text{BPP}$. (Hint: Use the result of Lemma 10.5.)

## Step-by-step solution

**Step 1** of 1

Given $M$ be probabilistic Turing Machine and $C$ be a language where for some fixed $0 < \varepsilon_1 < \varepsilon_2 < 1$,

1. $w \notin C$ implies $\Pr[M \text{ accepts } w] \leq \varepsilon_1$,
2. $w \in C$ implies $\Pr[M \text{ accepts } w] \leq \varepsilon_2$

It is required to show $C \in BPP$.

• Between any two distinct real numbers $\varepsilon_1 < \varepsilon_2$ there exists another real number that lies strictly between them. Thus to choose $c$ such that $\varepsilon_1 < c < \varepsilon_2$.

• Consider another machine $S$ which repeatedly runs $M$. Now $S$ accepts if the proportion of $M's$ acceptance is greater or equal to $c$, and $S$ rejects if the proportion of $M's$ acceptance is less than $c$. Now to show $S$ decides in $BPP$.

• Consider the variable $S_k$ be the total number of acceptances by machine $M$ after $k$ runs on input $w$. Hence, for $w \in C$, $S_k$ is the sum of $k$ $0-1$ random variables with common mean $\mu_2 > \varepsilon_2$, and for $w \notin C$, $S_k$ is sum of $k$ $0-1$ random variable with common mean $\mu_1 > \varepsilon_1$. The error probabilities can then be expressed as follows:

1.For $w \in C, \Pr[S \text{ rejects } w] = \Pr\left[\frac{S_k}{k} < c\right] \leq \Pr\left[\left|\frac{S_k}{k} - \mu_2\right| > \mu_2 - c\right]$

2.For $w \in C, \Pr[S \text{ accepts } w] = \Pr\left[\frac{S_k}{k} \geq c\right] \leq \Pr\left[\left|\frac{S_k}{k} - \mu_1\right| \geq c - \mu_1\right]$

By the weak law of large numbers (or various other bounds from probability theory), there exist $k$ that will make those probabilities on the right as small as desired, and in particular, there exist $k$ that will make them both strictly less than $\frac{1}{2}$.

• By using “Amplification lemma” this shows $C \in BPP$.

Comment

## Problem

Show that if P = NP, then P = PH.

## Step-by-step solution

**Step 1** of 1

It is required to show that if $P = NP,\ \text{then } P = PH$.

• Firstly, if $P = NP$, then because $P$ is closed under complement, thus $P = C_o NP$. Written as, $P = \sum_1 P = \prod_1 P$.

• Now using induction that if $P = \sum_i P = \prod_i P,\ \text{then } P = \sum_{i+1} P = \prod_{i+1} P$.

1. Assume $\sum_{i+1} P$ machine $M$ ,that consists of a run of the existential branching, then existential branching etc.

2. Assume the computation sub-tree path whose root are first universal step along path. For each such type of sub-tree, $M$ is performing a $\prod_i$ computation. By hypothesis, $\prod_i P = P$.

3. Thus for the forming of new machine $S$ each of computation sub-trees can be replaced by deterministic (non- branching) polynomial time of computation.

4. If assume $a(n)$ be the number of maximum steps which are taken by other machine before the start of universal machine, $P(n)$ be the maximum steps which are taken by any deterministic which were substituted for $\prod_i$ computations in $P$ machines, therefore $S$ is covered by $a(n) + p(a(n))$. Remember that the $p(a(n))$ term is composition of the functions, because $P$ sub procedures with inputs are computing which may be a longer than $n$ (but it must be equal or smaller than $a(n)$, since only $a(n)$ steps are executed on the time the sub procedures are used).

5. Since $a$ and $p$ both are polynomials, Therefore, $S$ is in $NP$. By hypothesis $P = NP$, so $S$ is in $P$ as well.

6. A similar type of argument may be used to reduce a $\prod_{i+1} P$ machine to $P = C_o NP$ machine , hence, putting it in $P$ as well, and completing collapse of hierarchy.

Comment

## Problem

Show that if PH = PSPACE, then the polynomial time hierarchy has only finitely many distinct levels.

## Step-by-step solution

**Step 1** of 2

The hierarchy in polynomial-time exists between deterministically accepted languages of classes $\mathrm{P}$ **in polynomial time** and deterministically or non-deterministically accepted languages of class $\mathrm{PSPACE}$ **in polynomial space**. There exists a relativization, which allowed minimum three levels of the hierarchy. The number of distinct levels, which are used to determine "low" and "high", in the hierarchy of polynomial-time are, depends upon the perishing of the $\mathrm{NP}$ class. Now consider the facts of "low" and "high", which is explained below:

• If there exist some $\mathrm{i}$ for which $\sum_{i}^{P}(Y) \subseteq \sum_{i}^{P}$ then a set $\mathrm{E}$ in $\mathrm{NP}$ is known as "low" and if there exists $\sum_{i+1}^{P} \subseteq \sum_{i}^{P}(z)$ for some $\mathrm{i}$, then it is known as "high".

Comment

**Step 2** of 2

Now, suppose $\mathrm{PH}$ **is defined as the union of the various classes of polynomial time hierarchy**. From the explanation of "low" and "high" (as it is defined above) , it can be shown that "**the hierarchy collapses are the only way of simultaneously existence of high and low for a set of PH.**

• There are two principle results exists. First one is based on the fact "either all or none (that is, every sparse set in $\mathrm{PH}$ is high or no one is low. Second one is that "every set of sparse will be extended high or no one sets will be extended high.

• Simply, it can be said that "**the hierarchy collapses in polynomial-time are** the only way for simultaneous existence of high and low behavior".

• A disjoint set can be obtained by combining high sets and low sets. The reason behind it is "the existence of immeasurably many levels extended by the hierarchy in polynomial time and in $\mathrm{NP}$ there exist some sets which show neither low nor high.

• **Therefore, the hierarchy in polynomial-size can be extended to only distinct finitely level if a sparse set in $\mathrm{NP}$.** Hence, it can be said that **if $\mathrm{PH=PSPACE}$, then the polynomial time hierarchy consists levels which are distinct and finite**.

Comment

## Problem

Recall that $NP^{SAT}$ is the class of languages that are decided by nondeterministic polynomial time Turing machines with an oracle for the satisfiability problem. Show that $NP^{SAT} = \Sigma_2 P$.

## Step-by-step solution

**Step 1** of 1

It is required to show $NP^{SAT} = \Sigma_2 P$.

1. Firstly, to show that $\Sigma_2 P \subseteq NP^{SAT}$.

• Suppose a language L is decided by $\Sigma_2$ alternating Turing machine $M$. Then that Turing machine performs some number of existential branching, followed by some number of universal branching.

• Consider the sub-trees of the computation path whose roots are the first universal step along the path. For each such sub-tree, $M$ is (by definition) performing a $\Pi_2$ computation. Thus, it is deciding a language in $Co\text{-}NP$ (Note: it may possibly be a different language for each sub-tree, but it does not matter).

• Now consider a nondeterministic machine $S$ that behaves just as $M$ does, but replaces each of the computation sub-trees discussed above with a deterministic computation by a machine in $P^{SAT}$ (this is feasible because $Co-NP \subseteq P^{SAT}$). Note that this machine only contains one run of existential branches, each extended with a deterministic computation that uses an $SAT$ oracle and runs in polynomial time.

• If assume $a(n)$ be the maximum number of steps taken by the alternating machine before the universal branches start, and $p(n)$ be the maximum number of steps taken by any of the $P^{SAT}$ machines which have substituted for the universal branching, then the running time of $S$ is bounded by $a(n)+p(a(n))$. Note that the $p(a(n))$ term is a composition of functions, because the $P^{SAT}$ sub procedures are computing with inputs that may be longer than $n$ (but must be smaller than or equal to $a(n)$, since only $a(n)$ steps have been executed at the time the sub-procedures are used).

• Since $a$ and $p$ are both polynomials, so is their composition. Therefore, $S$ is in $NP^{SAT}$.

2. Next, to show that $NP^{SAT} = \Sigma_2 P$.

• Suppose a language $L$ is decided by a $NP$ machine $M$ with an $SAT$ oracle.

• Consider the following modification of $M$. At each step of $M's$ computation that depends on the result of an oracle query, replace the step with a nondeterministic “split" that guesses the answer to the query. Each branch of the “split" also writes down the queried formula and answer obtained. Thus, at every “leaf" node of the computation tree (whenever the computation terminates), the tape contains a record of all queries and answers which lead to that leaf. Finally, in order to preserve the proper accepting behavior of $M$, replace each accepting “leaf" of $M$ with a computation that checks that all the oracle.

• This last computation can be done purely with a run of existential branches followed by a run of universal branches. First, all the positively answered queries are checked by running an NP machine that decides SAT. Second, all of the negatively answered queries are checked by running a $Co\text{-}NP$ machine that decides $UNSAT$.

• Finally, we check that the machine we have produced is indeed in $\Sigma_2 P$. Clearly, the steps have been ordered so that all of the existential branches precede all of the universal branches. Now it is just required to check that the maximum branch length is polynomial with respect to the input. There are at most a polynomial number of oracle queries and replacing each of them with a branch that writes the formula and query result only lengthens the branch by a polynomial amount. Thus the total branch lengthening caused by this replacement is only a polynomial times a polynomial. Since the “checking" step at the end is performed by a polynomial number of applications of $NP$ and $Co\text{-}NP$ machines (each run on a formula that is at most polynomial larger than the original input), the total time taken for this “checking" step is polynomial as well. Therefore, the machine is in $\Sigma_2 P$.

Comment

## Problem

Prove Fermat's little theorem, which is given in Theorem 10.6. (Hint: Consider the sequence $a_1, a_2, \ldots$. What must happen, and how?)

**THEOREM 10.6**

If $p$ is prime and $a \in \mathcal{Z}_p^+$, then $a^{p-1} \equiv 1 \pmod{p}$.

## Step-by-step solution

**Step 1** of 1

**Statement:** if $p$ be a prime and $a \in Z^+{}_p$ then $a^{p-1} \equiv 1 (\text{mod } p)$. Here $Z^+{}_p$ is defined as $Z^+{}_p = \{1, \ldots, p-1\}$ and $(p, a)$ is co-prime.

**Proof:** consider the following first $p-1$ positive multiple of $a$.

$$a, 2a, 3a, \ldots, (p-1)a$$

• As the **little Fermat's theorem** states "$(p, a)$ is co-prime (that is, $p$ is not exactly divisible by $a$)". Suppose $xa$ and $ya$ are taken in such a way that, the modulo $p$ of $xa$ and $ya$ are equal.

• Now, it can be said that $\mathbf{x = s} (\text{mod } p)$. So the $p-1$ multiples by $a$ above are non-zero and distinct; that is, they must be congruent to $a, 2a, 3a, \ldots, (p-1)a$ in the same order. Now, multiply **all these congruence together** and which gives:

$$a, 2a, 3a, \ldots, (p-1)a = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1) (\text{mod } p)$$

$$a^{p-1}(p-1)! = (p-1)! (\text{mod } p)$$

• Now, dividing each side by $(p-1)!$ in the above equality

$$a^{p-1} \equiv 1 (\textbf{mod } p)$$

**It is also known as a little Fermat's theorem and sometimes it can also be represented as**

$$a^p = a (\text{mod } p)$$

Comment

## Problem

Prove that for any integer p > 1, if p isn't pseudoprime, then p fails the Fermat test for at least half of all numbers in $Z^+_p$

## Step-by-step solution

**Step 1** of 1

It sufficient to prove that elements set in $Z_p^+$ that pass the Fermat test forms multiplicative subgroup of $Z_p^+$. Since the subgroup order divides the group order, if subgroup is a strict subgroup, it must contain at most half of elements of group.

To show that the set is a subgroup, it is required to show that it is nonempty and closed under the inverses and multiplication.

• First, the set is nonempty, since $1^{p-1} \equiv 1 \bmod \mathrm{p}$.

• If $a^{p-1} \equiv 1 \bmod p$, and $b^{p-1} \equiv 1 \bmod p$, then $(ab)^{p-1} \equiv a^{p-1}b^{p-1} \equiv 1 \bmod p$, which shows closure under multiplication.

• If $a^{p-1} \equiv 1 \bmod p$, then multiplying both sides of the equation by the $\left(a^{-1}\right)^{p-1}$ shows that $1 \equiv \left(a^{-1}\right)^{p-1} \bmod p$. Thus, the set is closed under inverses.

Hence, on the other side if $p$ is not pseudo prime then $p$ fails Fermat Test for at least half of number.

Comment

## Problem

Prove that if A is a language in L, a family of branching programs ($B_1$,$B_2$, . . .) exists wherein each $B_n$ accepts exactly the strings in A of length n and is bounded in size by a polynomial in n.

## Step-by-step solution

**Step 1** of 2

**A branching program** is defined as "**a directed acyclic graph** where the variables are used to label all the nodes except only two output nodes which is labeled $0$ and $1$". The **query nodes** are defined as all the nodes which are labeled by the variables. All the query nodes consists two outgoing edges, labeled as 0 and 1. Both output nodes doesn't consists outgoing edges.

Comment

**Step 2** of 2

Consider a language $A$ which takes an **input length** of $n$. A set of **branching programs** is taken in such a way that each branching program accepts exactly the strings in $A$ of length.

• Now, the **Merge-sort** can be implemented as a circuit in which the input length of language $A$ has taken as the nodes of the branching program.

• It is used to compare two bits after recursively dividing the given inputs in to half. The total time taken here (to divide the inputs into equal halves iteratively) is $\log n$.

• Consider the inputs can be called as $x_1, x_2$ and the outputs can be called as $y_1$. Now, **the action of the merge-sort algorithm** can be mimicked on an array. It can be implemented one step at position to be the $n$ input, $n/2$**-output sub-circuit**.

• Now, **a pass** can be implemented as the serial concatenation of steps, which has a size $n \log n$. Therefore, this gives a size $n * \log n = O(n \log n)$.

• Therefore, it can be said that "a language with an input length of $n$ can be computed in $O(n \log n)$ size circuits by using branching program.

**Hence from the above explanation it can be said that the language $A$ is in logarithmic space. In other words, the language the language $A$ is in $L$ .**

Comment

## Problem

Prove that if A is a regular language, a family of branching programs ($B_1$,$B_2$, . . .) exists wherein each $B_n$ accepts exactly the strings in A of length n and is bounded in size by a constant times n.

## Step-by-step solution

**Step 1** of 4

A formal language, which can be expressed using a regular expression, is called as a **regular language**. In other words, it can be defined as a language which is recognized by a finite automation. All the languages which are finite are regular.

• Now consider a **regular language** $A$**,** then a family of branching program $(B_1, B_2, ...)$ in which a string, of length $n$ in language $A$, is accepted by each $B_n$ and is **bounded by a fixed time** $n$ **in size.** It can be achieved by a way which is given below.

Comment

**Step 2** of 4

Now consider a branching program**. A branching program** is known as "**a directed acyclic graph** where labels of all the nodes are maintained by the variables, except for two output nodes which are labeled as 1 or 0.

• All the nodes **whose labels are maintained by the variables** are called **query nodes**.

• Every query nodes consists of two outgoing edges: one is labeled 1 and another one is labeled 0. Both output nodes doesn't consists outgoing edges.

Comment

**Step 3** of 4

So, from the definition of **branching program** and **regular language** $A$ as defined above "the n-input function or a finite regular language can be computed by a branching program that consist a constant $O(n)$ size.

• A **bubble-sort** can be implemented as a circuit $n$. A set of **branching programs** is taken in such a way that each branching program accepts exactly the strings in $A$ of length $n$.

• It is used to compare two bits and after comparing, reordering them if necessary is rather easy. The inputs can be called as $x_1, x_2$ and the outputs can be called as $y_1, y_2$.

• A sub-circuit can be written which accomplishes this as $y_1 = OR(x_1, x_2)$ and $y_2 = AND(x_1, x_2)$ . **This circuit contains a size of two**.

• Now, the action of the bubble-sort algorithm can be mimicked on an array. It can be implemented one step at position to be the $n$ input, $n$-output sub-circuit that passes through all the inputs taken as $< k\ and \geq k+1$ are unchanged.

• Now, the compare-swap sub-circuit, which is described above, on $< k\ and\ \geq k+1st$ input can be used to generate $kth$ and $k+1st$ output. This still has size two. Now, **a pass** can be implemented as the serial concatenation of steps for each of $k = 1, 2, ..., n-1$, which has a size $(n-1)*2$.

• A bubble-sort can be Proceed to implement as the serial concatenation of one passes. Therefore, this gives a size $\mathbf{1(n-1)*2 = O(n)}$.

Comment

**Step 4** of 4

Hence, from the above explanation it can be said that "**a family of branching program** $(B_1, B_2, ...)$ **in which a string, of length** $n$ **in language** $A$**, is accepted by each** $B_n$ **and is bounded by a fixed time** $n$ **in size if the language** $A$ **is regular**.

## Problem

Show that if NP $\subseteq$ BPP, then NP = RP.

## Step-by-step solution

**Step 1** of 2

As $RP \in NP$, it is sufficient to show $NP \subseteq RP$. It can be prove by showing that if $NP \subseteq BPP$, then $SAT \in RP$. Consider a formula $\phi$ with $n$ variables $x_1, x_2, \ldots\ldots x_n$ be the input. $\phi$ is satisfied iff $\exists$ truth assignment for $x_1, x_2, \ldots\ldots x_n$ so that $\phi(x_1, x_2 \ldots\ldots x_n) = 1$.

• Assume $A$ be a $BPP$ algorithm with error probability at most $2^{-k}$ for $SAT$, where $k = |\phi|$ is the length of formula $\phi$. Such $A$ exist because of the assumption that $SAT \in BPP$.

• First run $A$ on $\phi$. If $A$ rejects, then reject, otherwise try to satisfy assignment for $\phi$ on variable at a time.

• Initialize $x_1$ to $0$ and then call $A$ to determine that if the formula resulting is satisfiable: if $A$ returns "accept" then permanently set $x_1$ to $0$; otherwise set $x_1$ to $1$. Then proceed with $x_2$ similarly.

• If manage to construct a satisfying assignment at end then verify this assignment for $\phi$. If $\phi(x_1 \ldots, x_n) = 1$, then accept; otherwise reject.

• Here is the analysis. If $\phi$ is unsatisfiable, then always reject either because $A$ rejects in the process or do not arrive at a satisfying assignment at the end.

Comment

**Step 2** of 2

On the other hand, suppose $\phi$ is satisfiable. Proceed to show that it accept with probability at least $\frac{1}{2}$.

• Invoke $A$ a total of $n+1$ times. If $\phi$ is satisfiable and $A$ returns "accept" each time only for an assignment for variable $x_i$ which is part of a satisfying assignment, then end up with a satisfying assignment.

• Now show that the probability that at least one of the $n+1$ invocations returns "reject" for an assignment for variable $x_i$ which is part of a satisfying assignment is at most $\frac{1}{2}$.

• The probability that an invocation of $A$ returns does so is at most $2-k$. So probability that encounter it is at most $(n+1).2^{-k}$, which is at most $\frac{1}{2}$ because $n+1 \le k$.

Since both the algorithm $A$ and the construction of satisfying assignment run in polynomial time, the whole procedure clearly runs in polynomial time. Hence, $SAT \in RP$ and therefore $NP = RP$.

Comment

## Problem

Define a ZPP-***machine*** to be a probabilistic Turing machine that is permitted three types of output on each of its branches: *accept*, *reject*, and *?*. A ZPP-machine M decides a language A if M outputs the correct answer on every input string w

$$(accept \text{ if } w \in A \text{ and } reject \text{ if } w \notin A) \text{ with probability at least } \frac{2}{3}, \text{ and } M \text{ never}$$

outputs the wrong answer. On every input, M may output *?* with probability at most 1/3 . Furthermore, the average running time over all branches of M on w must be bounded by a polynomial in the length of w. Show that RP ∩ coRP = ZPP, where ZPP is the collection of languages that are recognized by ZPP-machines.

## Step-by-step solution

**Step 1** of 2

• $RP \bigcap co-RP \subseteq ZPP$. Consider $L \in RP \bigcap co-RP$.

Then assume $A$ be an $RP$ algorithm for $L$ and assume $B$ be an $co-RP$ algorithm for $L$.

• Consider $w$ be the input. One step of our $ZPP$ algorithm will be the following:

1. Run $A$ on $w$.

2. Then run $B$ on $w$ .

3. If $A$ accepts, accept.

4. If $B$ reject, reject.

Note that this step takes polynomial time since each $A$ and $B$ take polynomial time.

5. If $A$ rejects and $B$ accepts, repeat.

• First, show correctness.

Note that if $w \notin L$ then $A$ always rejects, so if $A$ accepts $w \in L$.

Similarly, if $B$ reject $w \notin L$. Thus, when output an answer, it is we always correct (which is better than probability $\frac{2}{3}$). Note that our algorithm never outputs $?^1$.

• Then, we prove that the running time is polynomial in expectation. Note that since each step has polynomial running time, it suffices to show the number of steps is polynomial in expectation (in fact, we will show it is constant in expectation). Note that if $w \in L$, then $A$ will accept with the probability at least $\frac{1}{2}$ (by definition of $RP$). Similarly, for $w \notin L$ and $B$ reject. Thus, every step succeeds with the probability of at least $\frac{1}{2}$ .

• Assume $X$ is a random variable denoting how many steps to take. Then let $p_k = \Pr[X = k]$ denote the probability by taking exactly $k$ steps $k-1$.

Note that $p_k$ is the probability that we fail on the first $k-1$ steps and then succeed, so

$$E|X| = \sum_{k=1}^{\infty} \frac{k}{2^{k-1}} = 4$$

And this is $\text{done}^2$.

Comment

**Step 2** of 2

$^1$In fact, $ZPP$ actually stands for zero-error probabilistic polynomial-time. This comes fact that it is equivalent to define $ZPP$ algorithms as never outputting a wrong answer.

$^2$General Formula can be drive to do last sum $(\text{for}|x|<1)$:

$$\begin{aligned}\sum_{k=1}^{\infty} kx^{k-1} &= \sum_{k=1}^{\infty}\left[\frac{d}{dx}x^k\right] \\ &= \frac{d}{dx}\left[\sum_{k=1}^{\infty} x^k\right] \\ &= \frac{d}{dx}\frac{1}{(1-x)} \\ &= \frac{1}{(1-x)^2}\end{aligned}$$

• $ZPP \subseteq RP$ (It can be proved that $ZPP \subseteq co-RP$ analogously) consider $L$ be in $ZPP$, and $A$ be an $ZPP$ algorithm for $L$, and consider $p(n)$ be the expected running time of $A$ on input of length $n$. Then build the following $RP$ algorithm:

• On input $w$ of length $n$,

1. Run $A$ on $w$ for time $6p(n)$.

2. If $A$ accepts or rejects, output the answer $A$.

3. Otherwise (If $A$ does not terminate, or outputs $?$), reject.

• Note that since $A$ has expected polynomial running time $p(n)$ is a polynomial, so $6p(n)$ is also polynomial. It suffices to prove correctness. First, if $w \notin L$, then $A$ will either reject, print $?$ or not terminate. In all of these cases, reject, so it is correct.

• On the other hand, assume $w \in L$. Again, three things can happen: $A$ accepts, $A$ prints $?$, or $A$ does not terminate in time. We are correct in the first two cases, so it suffices to show the probability of the latter two are at most $\frac{1}{2}$. The probability that $A$ outputs $?$ is at most $\frac{1}{3}$.

• The probability that $A$ does not terminate in 6p(n) time is at most $\frac{1}{3}$ (by Markov's inequality). Thus, if $w \in L$, the probability that $A$ is wrong is at most $\frac{1}{2}$, thus this algorithm is an $RP$ algorithm for $L$.

Comment

## Problem

Let $EQ_{BP}$ = { $\langle B_1, B_2 \rangle$ | $B_1$ and $B_2$ are equivalent branching programs}. Show that $EQ_{BP}$ is coNP-complete.

## Step-by-step solution

**Step 1** of 2

Consider $EQ_{BP} = \{\{B_1, B_2\} \mid B_1 \text{ and } B_2 \text{ both are equivalent branching program}\}$

Branching programs $B_1$ and $B_2$ can be described by acyclic graph rejects or may accept input strings $s_1, s_2, \ldots, s_n$. For turing machines these problems are un-decidable but these problems are $coNP$ complete for circuits.

Comment

**Step 2** of 2

Consider the problem $EQ_{BP}$ that is $B_1$ and $B_2$ is restricted to read-once programs. By using the equivalence with $coRP$ for testing the equivalence, and by reduction from $co\text{-}3SAT$ it will be $coNP$.

Polynomial can be determined by following way:

- Assign the vertex in programs $EQ_{BP}$ for branching, from root to final states.
- Label all incoming edges, now vertex polynomials will be sum of polynomials of edge which are incoming.
- Polynomial which is associated with final state $1$ will be branching program polynomial.

As the branching program is read-once, and have power not more than one. Hence polynomial cannot be more than degree of $n$. Hence, $EQ_{BP}$ must be $coNP$.

Comment

## Problem

Let BPL be the collection of languages that are decided by probabilistic log space Turing machines with error probability 1/3 . Prove that BPL $\subseteq$ P.

## Step-by-step solution

**Step 1** of 1

Suppose $\mathbf{BPL}$ be the collection of languages which are judged by **probabilistic log space** TM (Turing Machine) with an error probability of $1/3$. Now, suppose $L$ be a $\mathbf{BPL}$ language and **the machine** $M$ **is required** as the definition of $\mathbf{BPL}$ says.

• On input $x$ of length $n$, suppose the number of configuration of $M(.,x)$ is defined as $C$. A $C \times C$ matrix is constructed in such a way that $P[c_1, c_2] = 1/3$ if $c_2$ is reachable from $c_1$ in a single step, and $P[c_1, c_2] = 0$ otherwise.

• For all $t$, $P^t[c_1, c_2]$ is defined as the **probability of approaching configuration** $c_2$ **from configuration** $c_1$ in $t$ number of steps. Here, $P^t$ is defined as the matrix obtained by multiplying $P$ with itself $t$ times.

• The accepting probability of $M(.,x)$ can be **computed by** computing all powers of $P$ till the running time of $M(.,x)$ and **decide** if $x \in L$.

• The exact calculation can be performed at this time: **each probability is an integer multiple of** $1/3^{p(n)}$. **So, the polynomial number of digits can be used to represent it**.

Hence, it can be said that $\mathbf{BPL} \subseteq \mathbf{P}$.

Comment

## Problem

Let $CNF_H$ = { $\langle \phi \rangle \mid \phi$ is a satisfiable cnf-formula where each clause contains any number of literals, but at most one negated literal}. Problem 7.25 asked you to show that $CNF_H$ ? P. Now give a log-space reduction from *CIRCUIT VALUE* to $CNF_H$ to conclude that $CNF_H$ is P-complete.

## Step-by-step solution

**Step 1** of 4

Consider the following $CNF_H$ statement:

$CNF_H = \{ < \varnothing > | \varnothing$ is a satisfiable cnf-formula, where every clause consists any number of literals, but is consists maximum one negated literals $\}$

It is known that $\mathbf{CNF_H \in P}$.

Comment

**Step 2** of 4

Now, consider the **circuit evaluation** $CIRCUIT-VALUE$. For a circuit $C$ and input string $w$, the value of $C$ on $w$ can be written as $C(w)$. Then, $CIRCUIT-VALUE$ is given by

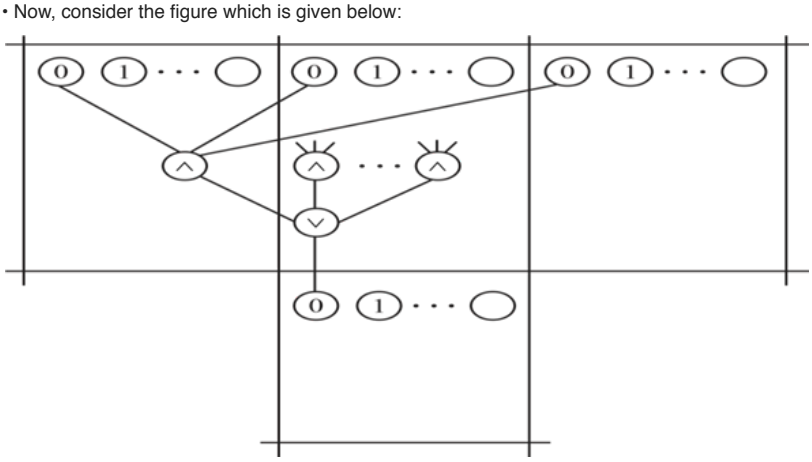
$$CIRCUIT-VALUE = \{\langle C, x \rangle \mid C \text{ is a Boolean circuit and } C(x) = 1\}$$

Comment

**Step 3** of 4

Consider the given theorem, which says that "**suppose** $t : M \to M$ **be a function, where** $t(m) \geq m$**. If** $W \in TIME(t(m))$**, then the complexity of the circuit** $A$ **is given by** $O(t^2(m))$

• Now, consider the figure which is given below:




• The above figure shows the construction of the theorem, which is discussed above. It shows the way of **reduction of a language** $W$ **(which is in** $P$**)** to $CIRCUIT-VALUE$.

• On input $w$, the production of a circuit takes place by the reduction. The process reduction simulates the Turing machine for $W$ in polynomial time. The $w$ itself can be taken as an input to the circuit.

• **A log-space is used to carry out the reduction because** the circuit produced by it contains a repetitive and a simple structure. **It shows that "** $CIRCUIT-VALUE$ **is** $P$**-complete.**

Comment

**Step 4** of 4

The above explanation **gives a log-space reduction from** $CIRCUIT-VALUE$ to $CNF_H$. Hence, from the above discussion it can be concluded that "$\mathbf{CNF_H}$ **is P - complete**".

Comment