Decidability:

The definition of an algorithm:

Algorithm is defined as a procedure which contains finite sequence of instructions that terminates after a finite number of steps for any input.

According to David Hilbert every definite mathe-matical problem must be susceptible for an exact settle-ment either in the form of an exact answer or by the proof of impossibility of its solution. He identified 23 mathematical problems as a challenge.

His 10th problem was to devise " A process according to which it can be determined by a finite number of operations", whether a polynomial over Z has an integral root

The formal definition of algorithm emerged after the works of Alan Turing and Alanzo church in 1936. The church-Turing thesis states that any algorithmic procedure that can be carried out by a human or a computer, can also be carried out by a Turing machine. Thus the Turing machine arose as an ideal theoretical model for an algorithm. Turing machine provided a machinery to mathematicians for attacking Hilbert's 10th problem.

The problem can be restated as: " does there exist a TM that can accept a polynomial over n variables if it has an integral root and reject the polynomial if it does not have one.

After the work of Yuri Matijasevic in 1970, it is universally accepted by computer scientists that Turing machine is a mathematical model of an algorithm.

Decidability

Languages accepted by the turing machine can be categorised as

(i) TM halts on all input string generated by language

(ii) TM that never halts on some input strings.

Definition 10.1:

A Language $L \subseteq \Sigma^*$ is recursively enumerable if there exists a TM M, such that $L = T(M)$

* Recursively Enumerable set is a set X for which we have a procedure to determine whether a given element belongs to X or not.

Definition 10.2

A Language $L \subseteq \Sigma^*$ is recursive if there exists a TM (M) that satisfies the following 2 conditions.

(i) If $w \in L$ then M accepts w and halts

(ii) If $w \notin L$ then M eventually halts, without reaching an accepting state.

D * Recursive: A set X is recursive if we have an algorithm to determine whether a given element belongs to X or not.

Definition 10.3

A problem with two answers (YES/NO) is decidable if the corresponding language is recursive. The Language L is also called decidable.

Decidable problem is also called solvable problem.

Definition 10.4

A problem/Language is undecidable if it is not decidable.

undecidable problem is also unsolvable problem.

Decidable Languages

Definition 10.5

$A_{DFA} = \{(B, w) \mid B \text{ accepts the input string } w\}$.

Theorem 10.1

A_{DFA} is decidable.

Proof: construct a TM that always halts and also accepts A_{DFA}. TM M is defined as follows:

(i) Let B be a DFA and w an input string (B, w) is an input for Turing Machine M.

(ii) Simulate B and input w in the TM M.

(iii) If the simulation ends in an accepting state of B, then M accepts w. If it ends in a non accepting state of B. then M rejects w.

It is evident that M accepts (B, w) if and only if w is accepted by the DFA B.

Definition 10.6.

$A_{CFG} = \{(G, w) \mid$ the context-free grammar G accepts input string $w\}$.

Theorem 10.2

A_{CFG} is decidable.

Proof: convert CFG into CNF. Then any derivation of w of length k requires 2k-1 steps if the grammar is in CNF.

Design a TM M that halts as follows:

(i) Let G be a CFG in CNF form and w an input string (G, w) is an input for M.

(ii) If k=0, List all the single step derivations. If k≠0, List all the derivation with 2k-1 steps

(iii) If any of the derivations in step-2 generates the given string w, M accepts (G, w), otherwise M rejects.

(G, w) is represented as (V_N, Σ, P, S) and input string w M accepts (G, w), if and only if w is accepted by the CFG, G.

Definition 10.7

$A_{CSG} = \{(G, w) \mid$ the context-sensitive grammar G accepts the input string $w\}$.

Theorem 10.3.

A_{CSG} is undecidable.

Proof:- In case of context-sensitive construct $W_i = \{\alpha \in (V_N \cup \Sigma)^* \mid S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ in i or fewer steps and $|\alpha| \leq n\}$. There exists a natural number K such that $W_K = W_{K+1} = W_{K+2} = \cdots$

So $w \in L(G)$ if and only if $w \in W_K$. The construction of W_K is the key idea used in the construction of TM accepting A_{CSG}.

Design a Turing Machine M as follows:

(i) Let G be a CSG and w an input string of length n. Then (G, w) is an input for TM.

(ii) construct $W_0 = \{S\}$, $W_{i+1} = W_i \cup \{\beta \in (V_N \cup \Sigma)^* \mid$ there exists $\alpha_i \in W_i$ such that $\alpha \Rightarrow \beta$ and $|\beta| \leq n\}$. continue until $W_k = W_{k+1}$ for some k.

(iii) If $w \in W_k$ $w \in L(G)$ and M accepts (G, w); otherwise M rejects (G, w).

Undecidable Languages.

Theorem 10.4

There exists a language over Σ_1 that is not recursively enumerable.

Proof: A language L is recursively enumerable if there exists a TM M such that $L = T(M)$. As Σ_1 is finite, Σ_1^* is countable.

As a turing machine M is a 7-tuple $(Q, \Sigma_1, \Gamma, \delta, q_0, b, F)$ and each member of the 7-tuple is a finite set. M can be encoded as a string. So the set I of all TM's is countable.

Let $\mathcal{L}$ be the set of all languages over Σ_1. Then a member of $\mathcal{L}$ is a subset of Σ_1^*.

we show that $\mathcal{L}$ is uncountable.

If $\mathcal{L}$ were countable then $\mathcal{L}$ can be written as sequence $\{L_1, L_2, \ldots\}$.

Σ_1^* can be written as sequence of $\{w_1, w_2, \ldots\}$

so L_i can be represented as an infinite binary sequence $x_{i1} x_{i2} x_{i3} \ldots$ where

$$x_{ij} = \begin{cases} 1 & \text{if } w_j \in L_i \\ 0 & \text{otherwise.} \end{cases}$$

using this representation L_i can be written as an infinite binary sequence

$$L_1 : x_{11} x_{12} x_{13} \ldots x_{1j} \ldots$$

$$L_2 : x_{21} x_{22} x_{23} \ldots x_{2j} \ldots$$

$$\vdots$$

$$L_i = x_{i1} x_{i2} x_{i3} \ldots x_{ij} \ldots$$

Let L be a subset of Σ_1^* by the binary sequence $y_1 y_2 y_3 \ldots$ where $y_i = 1 - x_{ij}$ If $x_{ij} = 0$ then $y_i = 1$ and if $x_{ij} = 1$ then $y_i = 0$. Thus the assumption L is a subset of Σ_1^* represented as binary sequence $y_1 y_2 y_3 \ldots$ should be L_k for some natural number k

But $L \neq L_k$ since $w \in L$ iff $w_k \notin L_k$. This contradicts the assumptions that $\mathcal{L}$ is countable. Hence $\mathcal{L}$ is uncountable.

As I is countable, $\mathcal{L}$ should have some members not corresponding to any TM in I. This proves the existence of a language over Σ that is not recursively enumerable.

Definition 10.8

$A_{TM} = \{(M, w) \mid \text{The TM } M \text{ accepts } w\}$.

Theorem 10.5

A_{TM} is undecidable.

Proof :- To prove A_{TM} is recursively enumerable, construct TM U as follows: (M, w) is an input to U. Simulate M on w. If M enters an accepting state, U accepts (M, w). Hence A_{TM} is recursively enumerable. Lets prove A_{TM} is undecidable by contradiction. Lets assume that A_{TM} is decidable by a TM H that eventually halts on all inputs then;

$$H(M, w) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w. \end{cases}$$

construct a new TM D with H as subroutine. D calls H to determine what M does when it recieves the input $\langle M \rangle$. Based on recieved information on $(M, \langle M \rangle)$; D rejects M if M accepts $\langle M \rangle$ and accepts $\langle M \rangle$ if M rejects $\langle M \rangle$. D is described as.

(i) $\langle M \rangle$ is an i/p to D, where $\langle M \rangle$ is the encoded string representing M.

(ii) D calls H to run on $(M, \langle M \rangle)$

(iii) D rejects $\langle M \rangle$ if H accepts $(M, \langle M \rangle)$ and accepts $\langle M \rangle$ if H rejects $(M, \langle M \rangle)$.

This means D accepts $\langle D \rangle$ if D does not accept $\langle D \rangle$ which is contradiction. Hence A_{TM} is undecidable.

Halting problem of Turing machine.

Reduction Technique is used to prove the undecidability of halting problem of Turing Machin

problem A is reducible to problem B if a sol to problem B can be used to solve problem A.

Example: Let problem A is to find the root o

$$x^4 - 3x^2 + 2 = 0.$$

problem B is to find the root of x^2-2

Then A is said to be reducible to B as.

(x^2-2) [problem-B] is a factor of x^4-3x^2+2

[problem-A]

IF A IS REDUCIBLE TO B & B IS DECIDABLE THEN A IS DECIDABLE. IF A IS REDUCIBLE TO B & B IS UNDECIDABLE THEN A IS ALSO UNDECIDABLE.

Theorem 10.6

$HALT_{TM} = \{(M, w) \mid$ The turing machine M halts on input $w\}$ is undecidable.

Proof: Assume that $HALT_{TM}$ is decidable.

Let M_1 be the TM such that $T(M_1) = HALT_{TM}$ and Let M_1 halt eventually on all (M, w).

construct TM M_2 as follows:

(i) For M_2, (M, w) is an input

(ii) The TM M_1 acts on (M, w)

(iii) If M_1 rejects (M, w) then M_2 rejects (M, w)

(iv) If M_1 accepts (M, w), simulate the TM M on the input string w until M halts

(v) If M has accepted w. M_2 accepts (M, w) other M_2 rejects (M, w).

In step (v) M_2 accepts (M, w). or M_2 rejects (M, w).

i.e $T(M_2) = \{(M, w) \mid$ TM accepts $w\}$.

$= A_{TM}$.

This is in contradiction; since A_{TM} is undecidable.

The Post Correspondance Problem [PCP]

PCP was first introduced by Emil Post in 1946. PCP is defined as the "problem over an alphabet Σ, belongs to a class of yes/no problems and is stated as follows";

consider two lists $x = (x_1, x_2 \dots x_n)$ and $y = (y_1, y_2, \dots y_n)$ of non empty strings over an $\Sigma = \{0, 1\}$. The PCP is to determine whether or not there exists $i_i \dots i_{np}$ where $1 \le i_j \le n$. such that $x_{i_1} \dots x_{im} = y_{i_1} \dots y_{im}$.

* indicies i need not be distinct.

Example:-

1. Does the PCP with two lists $x = (b, bab^3, ba)$ and $Y = (b^3, ba, a)$ have a solution?

solution:- Determine whether or not there exists a sequence of substrings of x such that the string formed by this sequence is identical to the string formed by the corresponding substrings of y are identical.

one of the sequence $= [i_1 = 2; i_2 = 1; i_3 = 1, i_4 = 3 \ (2, 1, 1, 3)]$ and $m = 4$.

i.e. $x = (\underbrace{b}_{x_1}, \underbrace{bab^3}_{x_2}, \underbrace{ba}_{x_3})$. $\quad y = (\underbrace{b^3}_{y_1}, \underbrace{ba}_{y_2}, \underbrace{a}_{y_3})$

$(2, 1, 1, 3) = [x_2, x_1, x_1, x_3]$ & $[y_2, y_1, y_1, y_3]$.

$\underline{bab^3}.\underline{b}.\underline{b}.\underline{ba}$ $\qquad$ $\underline{ba}.b^3.b^3.a$

$\underline{ba\ b^3\ b^3\ a}$ $\quad = \quad$ $\underline{ba}.b^3.\underline{b^3\ a}$

They are identical

Thus the PCP has a solution.

2. Prove that PCP with 2 lists $x = (01, 1, 1)$, $Y = (01^2$ has no solution.

Sol^: For each substring $x_i \in X$ and $Y_i \in Y$.
$|x_i| < |Y_i|$ for all i. Hence the string generc by a sequence of substrings of x is shorter than the string generealed by substrings of Y. Therefore PCP has no solution.

Explain how a PCP can be treated as a game o dominoes.

Sol^:

x_i
y_i

Let each domino contain some x_i in the upper-half & corresponding substring of Y in lower-half. PCP is equivalent to placing the dominoes one after the othe as a sequence. TO win the game, same string should appear in upper-half & lower-half. Thus winning the game is the solution to PCP.

Theorem 10.7.

The PCP over Σ for $|\Sigma| \geq 2$ is unsolvable.

Proof: The following results can be proved by the reduc-tion technique applied to PCP.

1. If L_1 and L_2 are any two CFL over an alphabet Σ and $|\Sigma| \geq 2$, there is no algorithm to determine whether or not
(a) $L_1 \cap L_2 = \phi$ (b) $L_1 \cap L_2$ is a CFL (c) $L_1 \subseteq L_2$ & (d) $L_1 = L_2$

2. If G is context sensitive grammar, there is no algorithm to determine whether or not
(a) $L(G) = \phi$ (b) $L(G)$ is infinile & (c) $x_0 \in L(G)$ for a fixed string x_0.

3. If G is a type 0 grammar, there is no algorithm to determine whether or not any string $x \in \Sigma^*$ is in $L(G)$.

Complexity

12.1 GROWTH RATE OF FUNCTIONS

When we have two algorithms for the same problem, we may require a comparison between the running time of these two algorithms. With this in mind, we study the growth rate of functions defined on the set of natural numbers.

In this section, N denotes the set of natural numbers.

Definition 12.1 Let $f, g : N \to R^+$ (R^+ being the set of all positive real numbers). We say that $f(n) = O(g(n))$ if there exist positive integers C and N_0 such that

$$f(n) \leq Cg(n) \qquad \text{for all } n \geq N_0.$$

In this case we say f is of the order of g (or f is 'big oh' of g).

Note: $f(n) = O(g(n))$ is not an equation. It expresses a relation between two functions f and g.

EXAMPLE 12.1

Let $f(n) = 4n^3 + 5n^2 + 7n + 3$. Prove that $f(n) = O(n^3)$.

Solution

In order to prove that $f(n) = O(n^3)$, take $C = 5$ and $N_0 = 10$. Then

$$f(n) = 4n^3 + 5n^2 + 7n + 3 \leq 5n^3 \qquad \text{for } n \geq 10$$

When $n = 10$, $5n^2 + 7n + 3 = 573 < 10^3$. For $n > 10$, $5n^2 + 7n + 3 < n^3$. Then, $f(n) = O(n^3)$.

Theorem 12.1 If $p(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$ is a polynomial of degree k over Z and $a_k > 0$, then $p(n) = O(n^k)$.

Proof $p(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$. As a_k is an integer and positive, $a_k \geq 1$.

As $a_{k-1}, a_{k-2}, \ldots, a_1, a_0$ and k are fixed integers, choose N_0 such that for all $n \geq N_0$ each of the numbers

$$\frac{|a_{k-1}|}{n}, \frac{|a_{k-2}|}{n^2}, \ldots, \frac{|a_1|}{n^{k-1}}, \frac{|a_0|}{n^k} \text{ is less than } \frac{1}{k} \qquad (*)$$

Hence,

$$\left|\frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \cdots + \frac{a_0}{n^k}\right| < 1$$

As $a_k \geq 1$, $\dfrac{p(n)}{n^k} = a_k + \dfrac{a_{k-1}}{n} + \cdots + \dfrac{a_1}{n^{k-1}} + \dfrac{a_0}{n^k} > 0 \qquad$ for all $n \geq N_0$

Also,

$$\frac{p(n)}{n^k} = a_k + \left(\frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}\right)$$

$$\leq a_k + 1 \qquad \text{by } (*)$$

So,

$$p(n) \leq Cn^k, \qquad \text{where} \qquad C = a_k + 1$$

Hence,

$$p(n) = O(n^k). \qquad \blacksquare$$

Corollary The order of a polynomial is determined by its degree.

Definition 12.2 An exponential function is a function $q : N \to N$ defined by

$$q(n) = a^n \qquad \text{for some fixed } a > 1.$$

When n increases, each of n, n^2, 2^n increases. But a comparison of these functions for specific values of n will indicate the vast difference between the growth rate of these functions.

TABLE 12.1 Growth Rate of Polynomial and Exponential Functions

n	$f(n) = n^2$	$g(n) = n^2 + 3n + 9$	$q(n) = 2^n$
1	1	13	2
5	25	49	32
10	100	139	1024
50	2500	2659	$(1.13)10^{15}$
100	10000	10309	$(1.27)10^{30}$
1000	1000000	1003009	$(1.07)10^{301}$

From Table 12.1, it is easy to see that the function $q(n)$ grows at a very fast rate when compared to $f(n)$ or $g(n)$. In particular the exponential function grows at a very fast rate when compared to any polynomial of large degree. We prove a precise statement comparing the growth rate of polynomials and exponential function.

Definition 12.3 We say $g \neq O(f)$, if for any constant C and N_0, there exists $n \geq N_0$ such that $g(n) > Cf(n)$.

Definition 12.4 If f and g are two functions and $f = O(g)$, but $g \neq O(f)$, we say that the growth rate of g is greater than that of f. (In this case $g(n)/f(n)$ becomes unbounded as n increases to ∞.)

Theorem 12.2 The growth rate of any exponential function is greater than that of any polynomial.

Proof Let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ and $q(n) = a^n$ for some $a > 1$.

As the growth rate of any polynomial is determined by its term with the highest power, it is enough to prove that $n^k = O(a^n)$ and $a^n \neq O(n^k)$. By L'Hospital's rule, $\dfrac{\log n}{n}$ tends to 0 as $n \to \infty$. (Here $\log n = \log_e n$.) If

$$z(n) = \left[e^{k\left(\frac{\log n}{n}\right)} \right]$$

then,

$$(z(n))^n = \left[e^{k\left(\frac{\log n}{n}\right)} \right]^n = e^{k \log n} = e^{\log n^k} = n^k$$

As n gets large, $k\left(\dfrac{\log n}{n}\right)$ tends to 0 and hence $z(n)$ tends to 0.

So we can choose N_0 such that $c(n) \leq a$ for all $n \geq N_0$. Hence $n^k = c(n)^n \leq a^n$, proving $n^k = O(a^n)$.

To prove $a^n \neq O(n^k)$, it is enough to show that a^n/n^k is unbounded for large n. But we have proved that $n^k \leq a^n$ for large n and any positive integer k and hence for $k + 1$. So $n^{k+1} \leq a^n$ or $\frac{a^n}{n^{k+1}} \geq 1$.

Multiplying by n, $n\left(\frac{a^n}{n^{k+1}}\right) \geq n$, which means $\frac{a^n}{n^k}$ is unbounded for large values of n. ∎

Note: The function $n^{\log n}$ lies between any polynomial function and a^n for any constant a. As $\log n \geq k$ for a given constant k and large values of n, $n^{\log n} \geq n^k$ for large values of n. Hence $n^{\log n}$ dominates any polynomial. But $n^{\log n} = (e^{\log n})^{\log n} = e^{(\log n)^2}$. Let us calculate $\lim_{x \to \infty} \frac{(\log x)^2}{cx}$. By L'Hospital's rule, $\lim_{x \to \infty} \frac{(\log x)^2}{cx} = \lim_{x \to \infty} (2 \log x) \frac{1/x}{c} = \lim_{x \to \infty} \frac{2 \log x}{cx} = \lim_{x \to \infty} \frac{2}{cx} = 0$.

So $(\log n)^2$ grows more slowly than cn. Hence $n^{\log n} = e^{(\log n)^2}$ grows more slowly than 2^{cn}. The same holds good when logarithm is taken over base 2 since $\log_2 n$ and $\log_2 n$ differ by a constant factor.

Hence there exist functions lying between polynomials and exponential functions.

12.2 THE CLASSES *P* AND *NP*

In this section we introduce the classes **P** and **NP** of languages.

Definition 12.5 A Turing machine M is said to be of time complexity $T(n)$ if the following holds: Given an input w of length n, M halts after making at most $T(n)$ moves.

Note: In this case, M eventually halts. Recall that the standard TM is called a deterministic TM.

Definition 12.6 A language L is in class **P** if there exists some polynomial $T(n)$ such that $L = T(M)$ for some deterministic TM M of time complexity $T(n)$.

EXAMPLE 12.2

Construct the time complexity $T(n)$ for the Turing machine M given in Example 9.7.

Solution

In Example 9.7, the step (i) consists of going through the input string (0^n1^n) forward and backward and replacing the leftmost 0 by x and the leftmost 1 by y. So we require at most $2n$ moves to match a 0 with a 1. Step (ii) is repetition of step (i) n times. Hence the number of moves for accepting a^nb^n is at most $(2n)(n)$. For strings not of the form a^nb^n, TM halts with less than $2n^2$ steps. Hence $T(M) = O(n^2)$.

We can also define the complexity of algorithms. In the case of algorithms, $T(n)$ denotes the running time for solving a problem with an input of size n, using this algorithm.

In Example 12.2, we use the notation $\leftarrow$ which is used in expressing algorithm. For example, $a \leftarrow b$ means replacing a by b.

$\lceil a \rceil$ denotes the smallest integer greater than or equal to a. This is called the *ceiling function*.

EXAMPLE 12.3

Find the running time for the Euclidean algorithm for evaluating gcd(a, b) where a and b are positive integers expressed in binary representation.

Solution

The Euclidean algorithm has the following steps:

1. The input is (a, b)
2. Repeat until $b = 0$
3. Assign $a \leftarrow a \bmod b$
4. Exchange a and b
5. Output a.

Step 3 replaces a by $a \bmod b$. If $a/2 \geq b$, then $a \bmod b < b \leq a/2$. If $a/2 < b$, then $a < 2b$. Write $a = b + r$ for some $r < b$. Then $a \bmod b = r < b < a/2$. Hence $a \bmod b \leq a/2$. So a is reduced by at least half in size on the application of step 3. Hence one iteration of step 3 and step 4 reduces a and b by at least half in size. So the maximum number of times the steps 3 and 4 are executed is $\min\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$. If n denotes the maximum of the number of digits of a and b, that is $\max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$ then the number of iterations of steps 3 and 4 is $O(n)$. We have to perform step 2 at most $\min\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$ times or n times. Hence $T(n) = nO(n) = O(n^2)$.

Note: The Euclidean algorithm is a polynomial algorithm.

Definition 12.7 A language L is in class **NP** if there is a nondeterministic TM M and a polynomial time complexity $T(n)$ such that $L = T(M)$ and M executes at most $T(n)$ moves for every input w of length n.

We have seen that a deterministic TM M_1 simulating a nondeterministic TM M exists (refer to Theorem 9.3). If $T(n)$ is the complexity of M, then the complexity of the equivalent deterministic TM M_1 is $2^{O(T(n))}$. This can be justified as follows. The processing of an input string w of length n by M is equivalent to a 'tree' of computations by M_1. Let k be the maximum of the number of choices forced by the nondeterministic transition function. (It is $\max\{|\delta(q, X)|\}$, the maximum taken over all states q and all tape symbol X.) Every branch of the computation tree has a length $T(n)$ or less. Hence the total number of leaves is atmost $kT(n)$. Hence the complexity of M_1 is at most $2^{O(T(n))}$.

It is not known whether the complexity of M_1 is less than $2^{O(T(n))}$. Once again an answer to this question will prove or disprove **P** $\neq$ **NP**. But there do exist algorithms where $T(n)$ lies between a polynomial and an exponential function (refer to Section 12.1).

12.8.1 QUANTUM COMPUTERS

We know that a bit (a 0 or a 1) is the fundamental concept of classical computation and information. Also a classical computer is built from an electronic circuit containing wires and logical gates. Let us study quantum bits and quantum circuits which are analogous to bits and (classical) circuits.

A quantum bit, or simply qubit can be described mathematically as

$$|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$$

The qubit can be explained as follows. A classical bit has two states, a 0 and a 1. Two possible states for a qubit are the states $|0\rangle$ and $|1\rangle$. (The notation $|\rangle$ is due to Dirac.) Unlike a classical bit, a qubit can be in infinite number of states other than $|0\rangle$ and $|1\rangle$. It can be in a state $|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$, where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. The 0 and 1 are called the computational basis states and $|\psi\rangle$ is called a superposition. We can call $|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$ a quantum state.

In the classical case, we can observe it as a 0 or a 1. But it is not possible to determine the quantum state on observation. When we measure/observe a qubit, we get either the state $|0\rangle$ with probability $|\alpha|^2$ or the state $|1\rangle$ with probability $|\beta|^2$.

This is difficult to visualize, using our 'classical thinking' but this is the source of power of the quantum computation.

Multiple qubits can be defined in a similar way. For example, a two-qubit system has four computational basis states, $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$ and quantum states $|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$ with $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$.

Now we define the qubit gates. The classical NOT gate interchanges 0 and 1. In the case of the qubit the NOT gate, $\alpha|0\rangle + \beta|1\rangle$, is changed to $\alpha|1\rangle + \beta|0\rangle$.

The action of the qubit NOT gate is linear on two-dimensional complex vector spaces. So the qubit NOT gate can be described by

$$\begin{bmatrix}\alpha\\ \beta\end{bmatrix} \rightarrow \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}\beta\\ \alpha\end{bmatrix}$$

The matrix $\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$ is a unitary matrix. (A matrix A is unitary if $A\,\text{adj}A = I$.)

We have seen earlier that {NOR} is functionally complete (refer to Exercises of Chapter 1). The qubit gate corresponding to NOR is the controlled-NOT or CNOT gate. It can be described by

$$|A, B\rangle \rightarrow |A, B \oplus A\rangle$$

where $\oplus$ denotes addition modulo 2. The action on computational basis is $|00\rangle \rightarrow |00\rangle$, $|01\rangle \rightarrow |01\rangle$, $|10\rangle \rightarrow |11\rangle$, $|11\rangle \rightarrow |10\rangle$. It can be described by the following 4×4 unitary matrix:

$$U_{CV} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Now, we are in a position to define a quantum computer:

A quantum computer is a system built from quantum circuits, containing wires and elementary quantum gates, to carry out manipulation of quantum information.

12.8.2 CHURCH–TURING THESIS

Since 1970s many techniques for controlling the single quantum systems have been developed but with only modest success. But an experimental prototype for performing quantum cryptography, even at the initial level may be useful for some real-world applications.

Recall the Church–Turing thesis which asserts that any algorithm that can be performed on any computing machine can be performed on a Turing machine as well.

Miniaturization of chips has increased the power of the computer. The growth of computer power is now described by Moore's law, which states that the computer power will double for constant cost once in every two years. Now it is felt that a limit to this doubling power will be reached in two or three decades, since the quantum effects will begin to interfere in the functioning of electronic devices as they are made smaller and smaller. So efforts are on to provide a theory of quantum computation which will compensate for the possible failure of the Moore's law.

As an algorithm requiring polynomial time was considered as an efficient algorithm, a strengthened version of the Church–Turing thesis was enunciated.

Any algorithmic process can be simulated efficiently by a Turing machine.

But a challenge to the strong Church–Turing thesis arose from analog computation. Certain types of analog computers solved some problems efficiently whereas these problems had no efficient solution on a Turing machine. But when the presence of noise was taken into account, the power of the analog computers disappeared.

In mid-1970s, Robert Solovay and Volker Strassen gave a randomized algorithm for testing the primality of a number. (A deterministic polynomial algorithm was given by Manindra Agrawal, Neeraj Kayal and Nitein Saxena of IIT Kanpur in 2003.) This led to the modification of the Church thesis.

G.1 Defining the Syntax of Programming Languages

Most programming languages are mostly context-free.1llere are some properties, such as type constraints. that cannot usually be described within the context-free framework. We will consider those briefly in Section 0.2. But context-free grammars provide the basis for defining most of the syntax of most programming languages.

G.1. 1 BNF

It became clear early on in the history of programming language development that designing a language was not enough. It was also necessary to produce an unambiguous language specification. Without such a specification. compiler writers were unsure what to write and users didn't know what code would compile. The inspiration for a solution to this problem came from the idea of a rewrite or production system as described years earlier by Emil Post. (See Section lR2.4.) In 1959. John Backus confronted the specification problem as he tried to write a description of the new language ALGOL 58. Backus later wrote {Backus 1980)"As soon as the need for precise description was noted it became obvious that Post's productions were well-suited for the purpose and I hastily adapted them for use in describing the syntax of IAL {Algol :58).

The notation that he designed was modified slightly in collaboration with Peter Naur and used in the definition. two years later, of ALGOL 60. The ALGOL 60 notation became known as BNF for Backus Naur form or Backus Normal form. For the definitive specification of ALGOL 60. using BNF. Just as the ALGOL 60 language influenced the design of generations of procedural programming languages. BNF has served as the basis for the description of those new languages, as well as others.

The BNF language that Backus and Naur used exploited these special symbols:

- ::= corresponds to →
- | means or. and
- < > surround the names of the nonterminal symbols.

EXAMPLE G.1 Standard BNF

Our term/factor grammar for arithmetic expressions would be written as follows in the original BNF language:

```
<E> ::= <E> + <T> | <T>
<T> ::= <T> * <F> | <F>
<F> ::= id | (<E>)
```

While it seems obvious to us now that formal specifications of syntax are important and BNF seems a natural way to provide such specifications, the invention of BNF was an important milestone in the development of computing. John Backus received the 1977 Turing Award for ··profound, influential. and lasting contributions to the design of practical high-level programming systems. notably through his work on FORTRAN. and for seminal publication of formal procedures for the specification of programming languages." Peter Naur received the 2005 Turing Award. For

fundamental contributions to programming language design and the definition of Algo\60,t o compiler design, and to the art and practice of computer programming."
Since its introduction in 1960, BNF has become the standard tool for describing the context-free part of the syntax of programming languages, as well as a variety of other formal languages: query languages, markup languages, and so forth. In later years. it has been extended both to make better use of the larger character codes that are now in widespread use and to make specifications more concise and easier to read. For example. Modern versions of BNF

- Use → instead of ::=
- Provide a convenient notation for indicating optional constituents. One approach is use subscripts $_{OPT}$ and another is to use [] to be metacharacters that surrounds optional constituent. The following rules iuustrate three ways to say the same thing:
  - ✓ S-> T|ε
  - ✓ S-> T_{OPT}
  - ✓ S->[T]

May include many of the features of RE, which are convenient fo specifying those parts of a language's syntax that do not require the full power of Context-free formalism.

These various dialects are called Extended BNF or EBNF

EXAMPLE G.2 EBNF

In standard BNF, we could write the following rule that describes the syntax of an identifier that must be composed of an initial letter, followed by zero or more alphanumeric characters:

```
<identifier> ::= <letter> | <letter> <alphanumseq>
<alphanumseq> ::= <alphanum> | <alphanum> <alphanumseq>
<alphanum> ::= <letter> | <digit>
```

ln EBNF, it can be written as:

```
identifier = letter (letter | digit)*
```

But note, this is a simple example that illustrate the point. In any practical system, the parsing of tokens, such as identifiers. is generally handled by a lexical analyzer and not by the context-free parser.

Applications: Security

J.1 Physical Security Systems as FSMs

Imagine a conventional intrusion-detection security system of the sort that is found in all kinds of buildings. including houses, offices, and banks. Such systems can naturally be modeled as finite-state machines. Some intrusion-detection systems are complex: They may. for example. divide the region that is being protected into multiple zones. Then the state of each zone may be partially or completely independent of the states or the other zones. But we can easily see the essential structure of such systems by considering the simple DFSM shown in Figure J. I. The inputs to this FSM arc user commands and timing events: arm (turn on the system). disarm (turn off the system). query the status of the system. reset the system. open a door, activate the glass-break detector, and 30 seconds' elapse. The job of this machine is to detect an intrusion. So, we have labeled the states that require an alarm as accepting states. State 6 differs from state 1 since it displays that an alarm has occurred since the last time the system was reset, and it will not allow the system to be armed until a reset occurs.

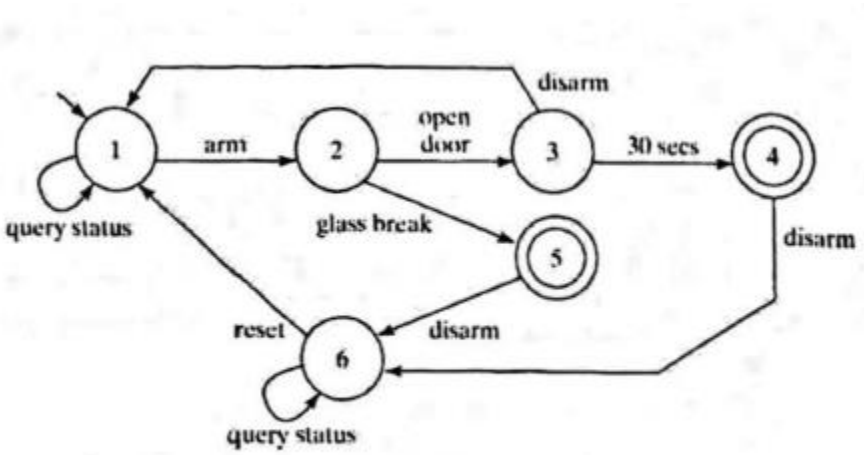


FIGURE J.1 A simple physical security system.

A realistic system has many more states. For example, suppose that alarm codes consist of four digits. Then the single transition from state 1 to state 2 is actually a sequence of four transitions, one for each digit that must be typed in order to arm the system. Suppose. for example, that the alarm code is 9999. Then we can describe the code-entering fragment of the system as the DFSM shown in Figure j.2.

Note that we have not specified what happens if the query button is pushed in states A-C. One of the questions that that system designer must answer is whether the query function is allowed in the middle of an arming sequence.

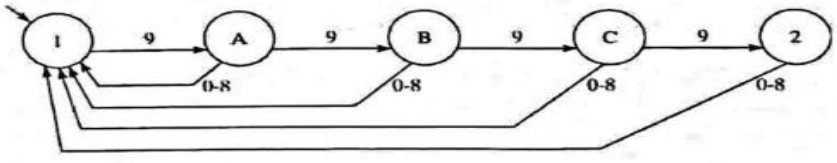


FIGURE J.2 The code-entering fragment of a security system.

J. 2 Computer System Safety

Consider a complex computer system. It includes files that some users, but not others, have access to. It includes processes (like print pay checks') that some users are allowed to run but most are not. Is it decidable whether such a system is safe? For example is it decidable. given the operations that are possible in the system. whether an unauthorized user could acquire access to the paycheck printing system'? The answer to this question depends, of course, on the operations that are allowed.

To build a model of the protection status. of a system. we'll use three kinds of entities:

- Subjects: Active agents, generally processes or users.
- Objects: Resources that the agents need to exploit. These could include files, processes, Devices,etc.). Notice that processes can be viewed both as subjects (entities capable of doing things) and as objects (entities that other entities may want to invoke),
- Rights: Capabilities that agents may have with respect to the objects. Rights could include read access, write access, delete access or execute access for files. execute access for processes, edit, compile, or execute access for source code check or change access for a password file, and so forth.

	$process_1$	$process_2$	$process_3$	$process_4$	$file_1$	$file_2$
$process_1$	execute		execute		own	read
$process_2$		execute	execute			read
$process_3$			execute			own
$process_4$			execute	execute		read

FIGURE J.3 A simple access control matrix.

We can describe the current protection status of a system with an access control matrix. A, that contains one row for each agent and one column for each protected object. Each cell of this matrix contains the set of rights that the agent possesses with respect to the object. Figure J.3 shows a simple example of such a matrix.

The protection status of a system must be able to evolve along with the system. We'll assume the existence of the following primitive operations for changing the access matrix:

- Create subject (x) records the existence of a new subject x, such as a new user or process.
- Create object (x) records the existence of a new object x. such as a process or a file.
- Destroy subject (x).
- Destroy object (x).
- Enter r into A[s, o] gives subjects the right r with respect tu object o,
- Delete r from A[s. o] removes subject s's right r with respect to object o.

We will allow commands to be constructed from these primitives, but all such commands must be of the following restricted form:

```
command-name(x1, x2, ..., xn) =
  if r1 in A[..., ...] and
    r2 in A[..., ...] and
       ...
    rj in A[..., ...]
  then
    operation1
    operation2
       ...
    operationm
```

ln other words. the command may check that particuh1r rights arc present in selected cells of the access matrix. If all conditions are met. then the operation sequence is executed. All the operations must be primitive operations as defined above. So no additional tests, loops. or branches are allowed. The parameters x., .t2 •• • •• • t, must each be bound to some subject or some object. The rights r1, r2,....ri are hard-coded into the definition of a particular command.

Define a protection framework to be, a set of commands that have been defined as described above and that are available for modifying an access control matrix. Define a protection system to be a pair (init, framework).

init is an initial configuration that is described by an access control matrix that contains various rights in its cells.

Framework is a protection framework that describes the way in which the rights, contained in the matrix can evolve as a result of system events.

In designing a protection framework, our goal is typically to guarantee that certain subjects maintain control over certain rights to certain objects. We will say that a right has leaked iff it is added to some access control matrix cell that did not already contain it. We will say that a protection system is safe with respect to some right riff there is no sequence of commands that could. if executed from the system's initial configuration, cause *r* to be leaked. We'll say that a system is *unsafe* iff it is not safe. Note that this definition of safety is probably too strong for most real applications. For example, if a process creates a file it will generally want to assign itself various rights to that file. That assignment of rights should not constitute leakage. It may also choose to allocate some rights to other processes. What it wants to be able to guarantee is that no further transfer of unauthorized rights will occur. That more narrow definition of leakage can be described in our framework in a couple of ways, including the ability to ask about teak~ age from an arbitrary point in the computation (e.g .. after the file has been created and assigned initial rights) and the ability to exclude some subjects (i.e .• those who are .. trusted") from the matrix when leakage is evaluated. For simplicity, we will consider just the basic model here.

• Given a protection system S = (init, framework) and a right r. is it decidable whether S is safe with respect to r?

It turns out that if we impose an additional constraint on the form of the commands in the system then the answer is yes. Define a protection framework to be mono-operational ·iff the body of each command contains a single primitive operation. The safety question for mono-operational protection systems is decidable. But such systems are very limited. for example, they do not allow

the definition of a command by which a subject creates a file and then gives itself some set of rights to that file. So we must consider the question of decidability of the more general safety question.

Given an arbitrary system S = (init, framework) and a right r. is it decidable whether S is safe with respect to r? Now the answer is no, which we can prove by reduction from H_T = {<M>:Turing machine M halts on ε}.

The key ideas in the proof are the following:

• it is possible to encode the configuration of an arbitrary Turing machine Mas an access control matrix we'll call A. To do this will require a set of "rights" as follows:

• One for each element of M's tape alphabet.

• One for each state of M. These must be chosen so that there is no overlap in names with the ones that correspond to tape alphabet symbols. Let q_f be the "right" that corresponds to any of M's halting states...

We call these objects "rights" (in quotes) because, although we will treat them like rights in a protection system, they are not rights in the standard sense. They do not represent actions that an agent can take. They are simply symbols that will be manipulated by the reduction. Each square of M's tape that is either non blank or has been visited by M will correspond to one cell in the matrix A . The cell that corresponds to $square_i$ of M's tape will contain the "right" that corresponds to the current symbol on $square_i$ of the tape. In audition. the matrix will encode the position of M's read/write head and its state. It will do that by containing in the cell that is currently under the read/write head the "right" that corresponds to M's current state.

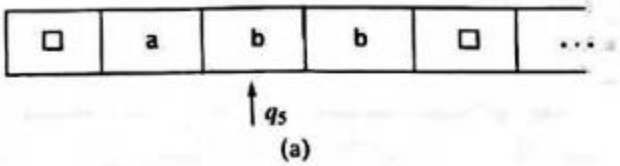
• It is possible to describe the transition function of a Turing machine as a protection framework (a set of commands as described above, for manipulating the access control matrix).

• So the question ... Does M ever enter one of its halting. states when started with an empty tape'?" can he reduced to the question. "If A starts out representing M's initial configuration. does a symbol corresponding to any halting state ever get inserted into any edit of A']" In other words, "Has any halting state symbol leaked?"

So. if we could decide whether an arbitrary protection system is safe with respect to an arbitrary right r. we could decide H_ε. But we know from Theorem 21.1. that H_ε is not in D.

The only question we are asking about M is whether it halts. If it halts, we don't care which of its halting states it lands in. So we will begin by modifying M so that it has a single halting state q_f. The modified M will enter q_f iff the original M would enter any of its halting states. Now we can ask the specific question. "Does q_f leak?" To make it easier to represent M's configuration as an access control matrix, we will assume that M has a one-way (to the right) infinite tape rather than our standard, two~ way infinite tape. By theorem 17.5. any computation by a Turing machine with a two way infinite tape can be simulated by a Turing machine with a one-way infinite tape, so this assumption does not limit the generality of the result that we are about to present. To see how a configuration or M is encoded as an access control matrix. consider the simple example shown in Figure J.4 (a). M is in state q~ and we assume that it started on the blank just to the left of the beginning of the input. so there are four nonblank or examined squares on M's tape. This

configuration will be represented as the square access control matrix A, shown in Figure J.4 (h). A contains one row and one column for each t



(a)

	s_1	s_2	s_3	s_4
s_1	□	*own*		
s_2		a	*own*	
s_3			b, q_5	*own*
s_4				b, *end*

(b)

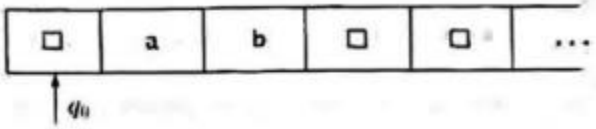
FIGURE J.4 Representing a Turing machine configuration as an access control matrix.

mimic the moves that *M* can make. For example, suppose that, in state q_5 reading b, *M* writes an a, moves left, and goes to state q_6. We construct the following command:

```
stateq5readingb(x1, x2) =
   if own in A[x1, x2] and          /* This command can only apply to
                                       two adjacent tape squares,
                                       where
      q5 in A[x2, x2] and           /* the one to the right is
                                       currently under the read/write
                                       head and M is in q5, and
      b in A[x2, x2]                /* there is a b under the
                                       read/write head
   then
      delete q5 from A[x2, x2]      /* Remove the old state info
      delete b from A[x2, x2]       /* and the current symbol under
                                       the read/write head.
      enter a into A[x2, x2]        /* Write the new symbol under the
                                       read/write head.
      enter q6 into A[x1, x1]       /* Move the read/write head one
                                       square to the left and go to
                                       state q6.
```

We must construct one such command for every transition of M. We must also construct commands that correspond to the special cases in which M tries to move off the tape to the left and in which it moves to the right to a previously unvisited blank tape square. The latter condition occurs whenever M tries to move right and the current tape square has the "right" end. In that case. the appropriate command must first create a new object and a new subject corresponding to the next tape square. The simulation of a Turing machine M begins by encoding M's initial configuration as an access control matrix. For example, suppose that M's initial configuration is as shown in Figure J.5(a). Then we let A be the access control matrix shown in Figure J.5(b). There

are a few other details that we must consider. For example, since we are going to test whether q/ever gets inserted into A during a computation, we must be sure that



(a)

	s_1	s_2	s_3
s_1	□, q_0	*own*	
s_2		a	*own*
s_3			b, *end*

(b)

FIGURE J.5 Encoding an initial configuration as an access control matrix.

qf, is not in A in the initial configuration. So if M starts in qf ; we will first modify it so that it starts in some new state and then makes one transition to t/ ~ Notice that we have constructed the commands in such a fashion that. if M is deterministic. exactly one command will have its conditions satisfied at any point. If M is nondeterministic then more than one command may match against some configurations. We can now show that it is undecidable whether given an arbitrary protection system S = (init, framework) and right r. S is safe with respect to r. To do so, we define the following language and show that it is not in D:

• Safety = { : S is safe with respect to r}

THEOREM J.1 "Is *S* is Safe with Respect to *r*?" is Undecidable

Theorem: The language Safety = $\{<S, r> : S \text{ is safe with respect to } r\}$ is not in D.

Proof: We show that $H_\varepsilon = \{<M> : \text{Turing machine } M \text{ halts on } \varepsilon\} \leq$ Safety and so Safety is not in D because H_ε isn't. Define:

$R(<M>)$ =

1. Make any necessary changes to M:
   1.1. If M has more than one halting state, then add a new unique halting state q_f and add transitions that take it from each of its original halting states to q_f.
   1.2. If M starts in its halting state q_f, then create a new start state that simply reads whatever symbol is under the read/write head and then goes to q_f.
2. Build S:
   2.1. Construct an initial access control matrix A that corresponds to M's initial configuration on input ε.
   2.2. Construct a set of commands, as described above, that correspond to the transitions of M.
3. Return $<S, q_f>$.

{R ¬} is a reduction from Hε to Safety. If Oracle exists and decides Safety then C= ¬ Oracle(R(<M>)) can be implemented as a Turing machines. And C is correct. By definition, S is unsafe with respect to qf iff qf is not present in the initial configuration of A and there exists some sequence of commands in S that could result in the initial configuration of S being transformed into a new configuration in which qf has leaked. i.e., it appears in some cell of A. Since the initial configuration of S corresponds to M being in its initial configuration on a blank tape, M does not start in qf, and the commands of S simulate the moves of M. this will happen iff M reaches state q1and so halts. Thus:

• If <M> ∈ Hε: M halts one ε, so qf eventually appears in some cell of A. S is unsafe with respect to qf, so Oracle rejects. C accepts.

• If <M> ∈ Hε :M does not halt on W, so qf, never appears in some cell of A. S is safe with respect to qf, so Oracle accepts. C rejects.

But no machine to decide Hε can exist, so neither does Oracle. Does the undecidability of Safety mean that we should give up on proving that systems are safe? No. There are restricted models that are decidable. And there are specific instances of even the more general model that can be shown to have specific properties. This result just means that there is no general solution to the problem.