

# Unit 2

# Syllabus

Outcomes for Unit I

Unit II	Analysis of Algorithms and Complexity Theory	07 Hours
<p>Analysis: Input size, best case, worst case, average case Counting Dominant operators, Growth rate, upper bounds, asymptotic growth, <math>O</math>, <math>\Omega</math>, <math>\Theta</math>, <math>o</math> and <math>\omega</math> notations, polynomial and non-polynomial problems, deterministic and non-deterministic algorithms, P-class problems, NP-class of problems, Polynomial problem reduction NP complete problems- vertex cover and 3-SAT and NP hard problem - Hamiltonian cycle.</p>		
#Exemplar/Case Studies	Analysis of iterative and recursive algorithm	

# Running Time

“What is the running time of your program?”

- Measuring running time like this raises so many other questions like:-
  - What’s the speed of the processor of the machine the program is running on?
  - What is the size of the RAM?
  - What is the programming language?
  - How experience and skillful the programmer is?  
and much more

- “My program runs in 3 seconds on Intel Core i7 8-cores 4.7 GHz processor with 16 GB memory and is written in C++ 14”.
- Now the question is how should we represent the running time so that it is not affected by the speed of computers, programming languages, and skill of the programmer?
- In another word, how should we represent the running time so that we can abstract all those dependencies away?.
- The answer to the question is simple which is “input size”.
- If the input size is  $n$  (which is always positive), then the running time is some function  $f$  of  $n$ . i.e. Running Time= $f(n)$

# Input Size

- Input size informally means the number of instances in the input.
- For example, if we talk about sorting, the size means the number of items to be sorted.
- If we talk about graphs algorithms, the size means the number of nodes or edges in the graph.

# Types of Analysis

- *The field of computer science, which studies efficiency of algorithms, is known as **analysis of algorithms**.*
- The function  $f(n)$ , gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data.
- The complexity function  $f(n)$  for certain cases are:
  1. **Best Case** : The minimum possible value of  $f(n)$  is called the best case.
    - Provides a lower bound on running time
    - Input is the one for which the algorithm runs the fastest
    - In the linear search problem, the best case occurs when  $x$  is present at the first location.
    - The number of operations in the best case is constant (not dependent on  $n$ ). So time complexity in the best case would be  $\Omega(1)$

# Types of Analysis

2. **Worst Case** : The maximum value of  $f(n)$  for any key possible input.
- Provides an upper bound on running time
  - An **absolute guarantee** that the algorithm would not run longer, no matter what the inputs are
  - For Linear Search, the worst case happens when the element to be searched ( $x$ ) is not present in the array. When  $x$  is not present, the `search()` function compares it with all the elements of `arr[]` one by one. Therefore, the worst-case time complexity of the linear search would be  $O(n)$ .

# Types of Analysis

**3. Average Case** : In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs.

- Sum all the calculated values and divide the sum by the total number of inputs.
- We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of  $x$  not being present in the array).
- The average case analysis is not easy to do in most practical cases and it is rarely done



# Step Count

Some basic assumptions are;

- There is no count for { and } .
- Each basic statement like 'assignment' and 'return' have a count of 1.
- If a basic statement is iterated, then multiply by the number of times the loop is run.
- The loop statement is iterated  $n$  times, it has a count of  $(n + 1)$ . Here the loop runs  $n$  times
- For the true case and a check is performed for the loop exit (the false condition), hence the additional 1 in the count

## 1. Sum of elements in an array

	Step-count (T.C)	Step-count (Space)
Algorithm Sum(a,n)		
{	0	
sum = 0;	1	1 word for sum
for i = 1 to n do	n+1	1 word each for i and n
sum = sum + a[i];	n	n words for the array a[]
return sum;	1	
}	0	
Total:	2n+3	(n+3) words

## 2. Adding two matrices of order $m$ and $n$

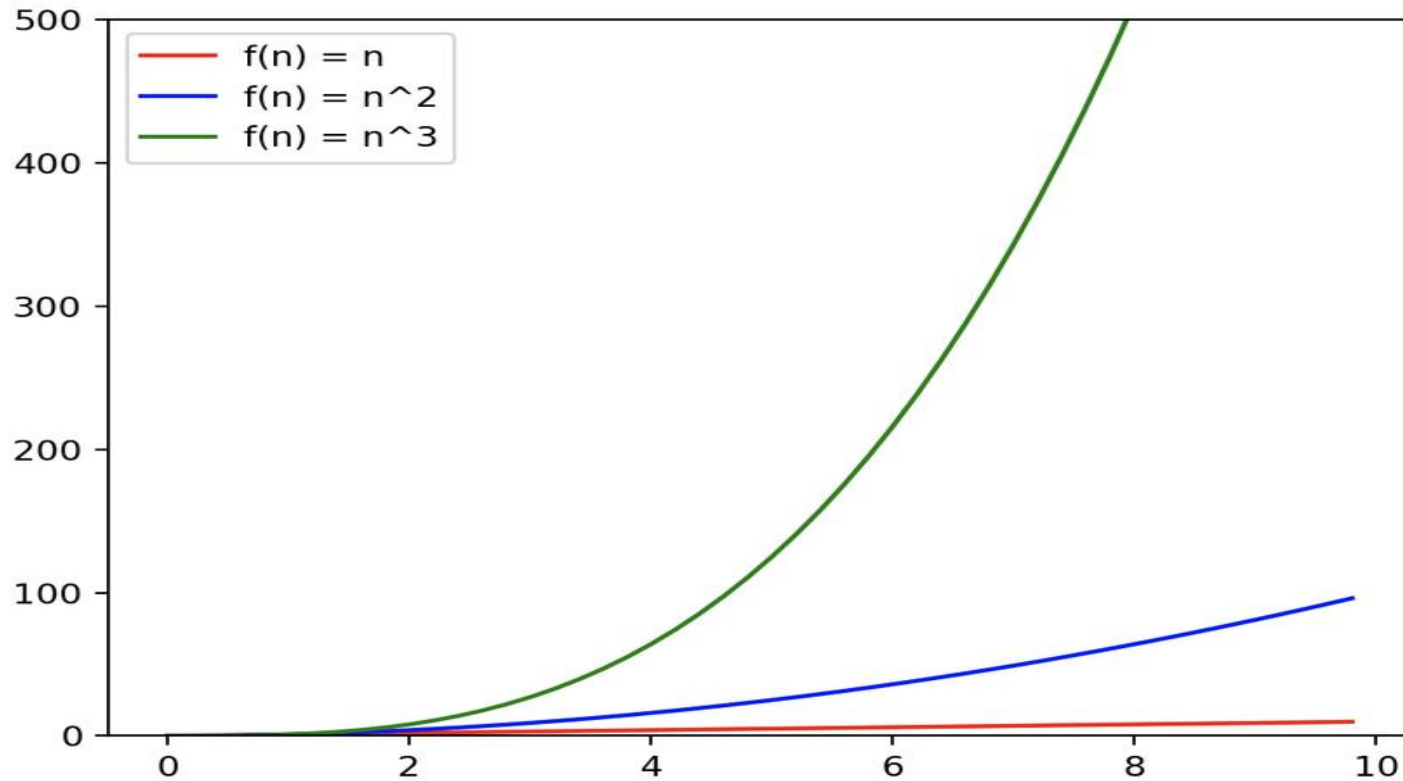
Algorithm Add(a, b, c, m, n)	Step Count
{	
for i = 1 to m do	---- m + 1
for j = 1 to n do	---- m(n + 1)
c[i,j] = a[i,j] + b[i,j]	---- m.n
}	-----
Total no of steps=	2mn + 2m + 2

Note that the first 'for loop' is executed  $m + 1$  times, i.e., the first  $m$  calls are true calls during which the inner loop is executed and the last call  $(m + 1)^{th}$  call is a false call.

# Growth of Functions

- Running times are expressed in terms of the input size ( $n$ ).
- Three possible running times are  $n$ ,  $n^2$  and  $n^3$ . Which is better
- To find this out, we need to analyze the growth of the functions i.e we want to find out, if the input increases, how quickly the running time goes up.

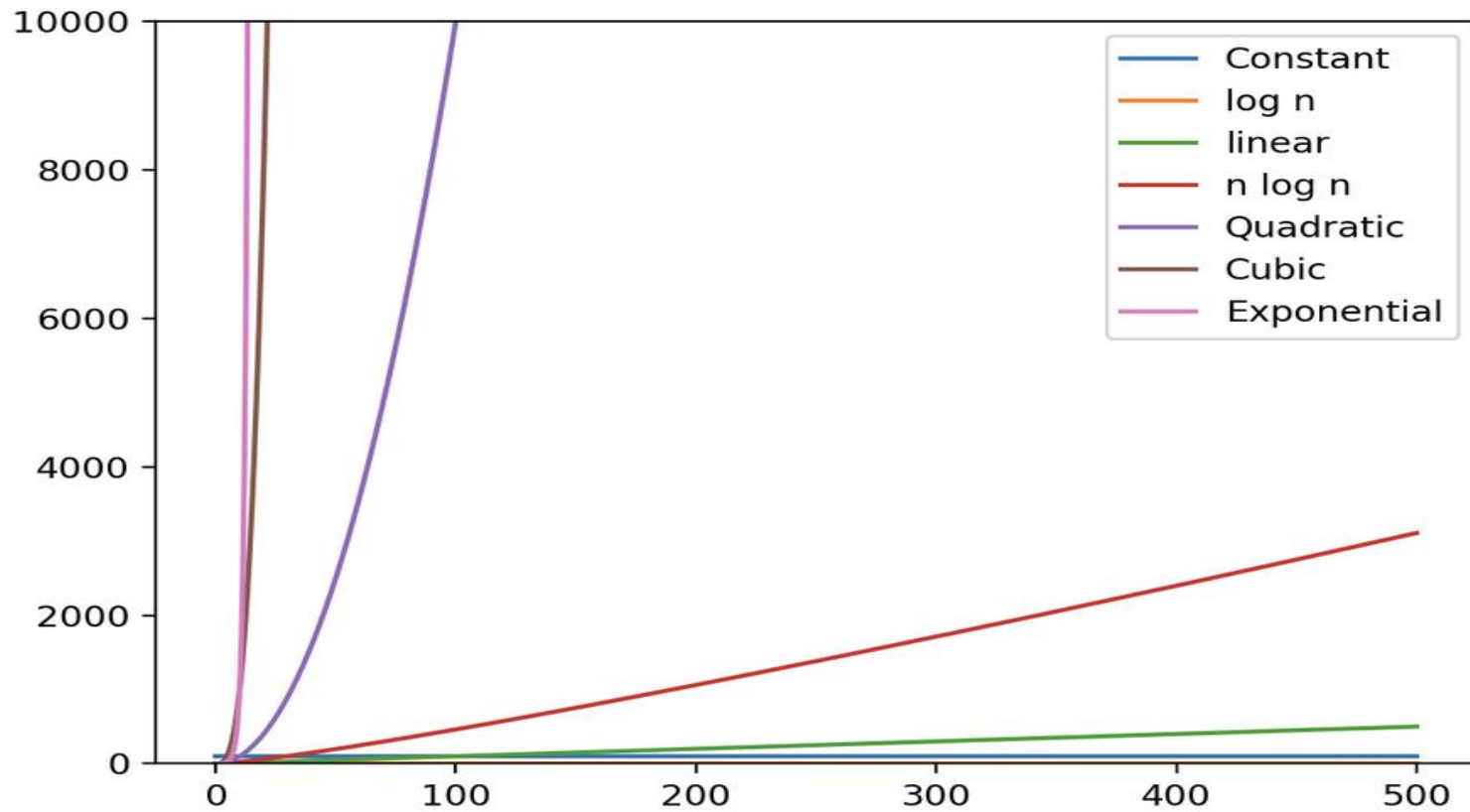
# Growth of Functions



# Growth of Functions

Running Time	Examples
Constant	1, 2, 100, 300, ...
Logarithmic	$\log n$ , $5 \log n$ , ...
Linear	$n$ , $n+3$ , $2n+3$ , ...
$n \log n$	$n \log n$ , $2n \log n + n$ , ...
Polynomial	Quadratic, Cubic, or higher order
Exponential	$2^n$ , $3^n$ , $2^n + n^4$ , ...
Factorial	$n!$ , $n! + n$ , ...

# Growth of Functions



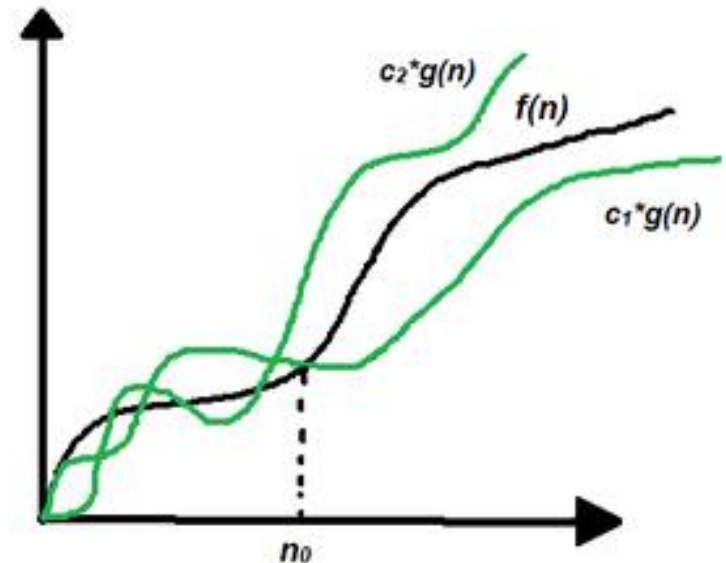
# Asymptotic Notations

The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:

- Big-O Notation ( $O$ -notation)
- Omega Notation ( $\Omega$ -notation)
- Theta Notation ( $\Theta$ -notation)

# $\Theta$ Notation

- Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.
- Let  $g$  and  $f$  be the function from the set of natural numbers to itself. The function  $f$  is said to be  $\Theta(g)$ , if there are constants  $c_1, c_2 > 0$  and a natural number  $n_0$  such that  $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$  for all  $n \geq n_0$



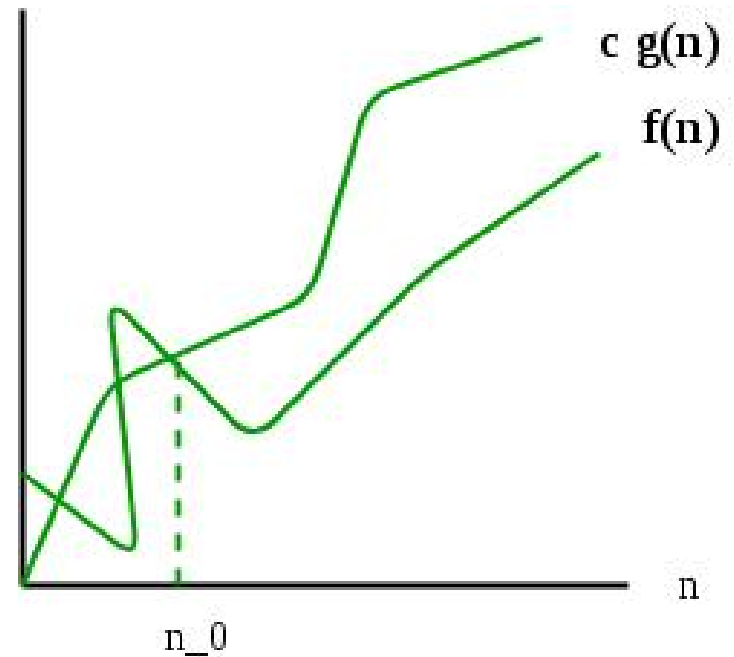


# $\Theta$ Notation

- Consider the expression  $3n^3 + 6n^2 + 6000 = \Theta(n^3)$ , the dropping lower order terms is always fine because there will always be a number  $(n)$  after which  $\Theta(n^3)$  has higher values than  $\Theta(n^2)$  irrespective of the constants involved. For a given function  $g(n)$ , we denote  $\Theta(g(n))$  is following set of functions.
- $\Theta$  provides exact bounds.

# Big O Notation:

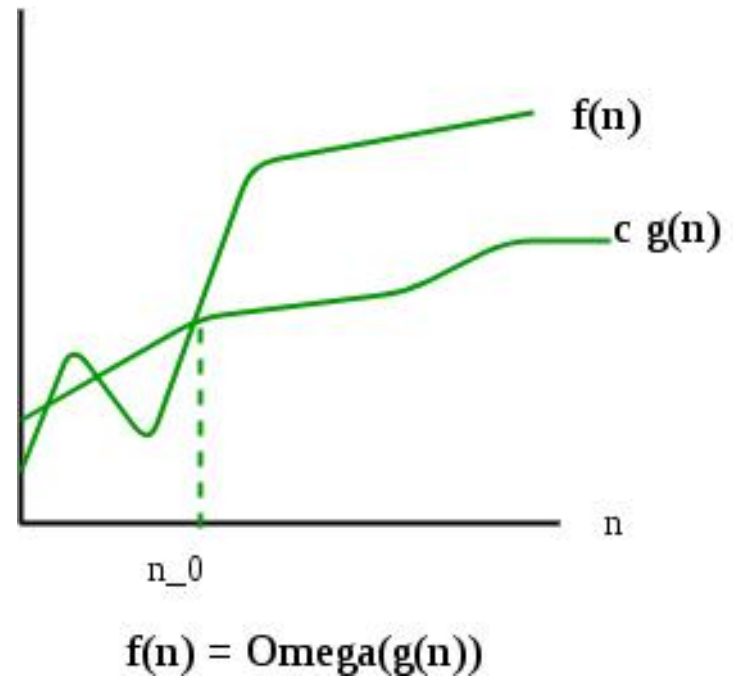
- Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.
- If  $f(n)$  describes the running time of an algorithm;  $f(n)$  is  $O(g(n))$  if there exist positive constant  $C$  and  $n_0$  such that,  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$
- Consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of the Insertion sort is  $O(n^2)$ .



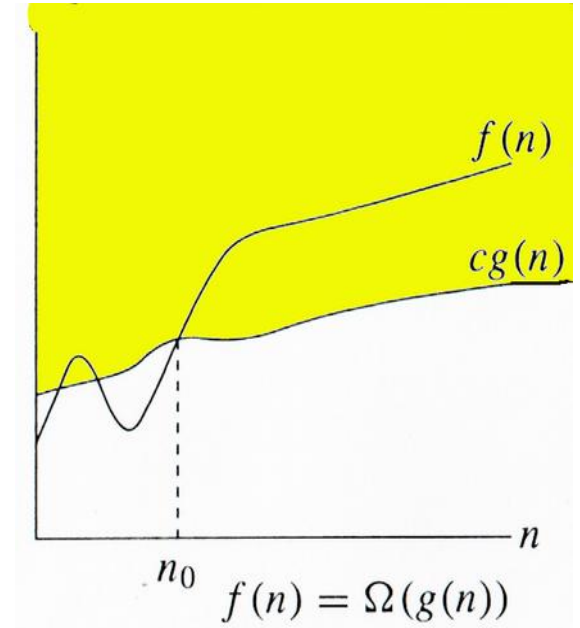
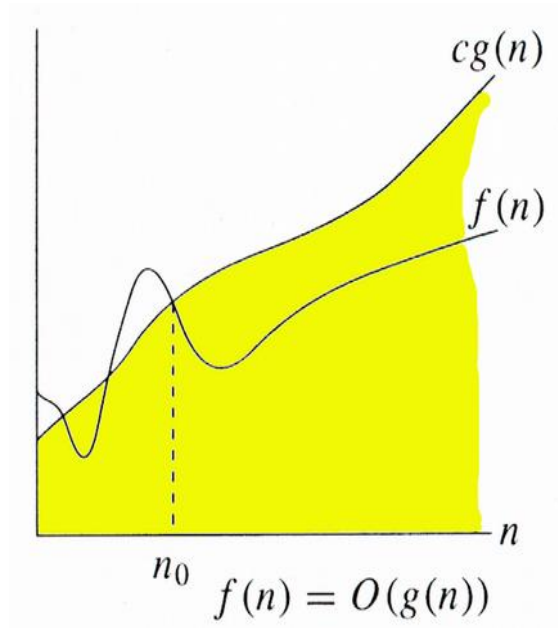
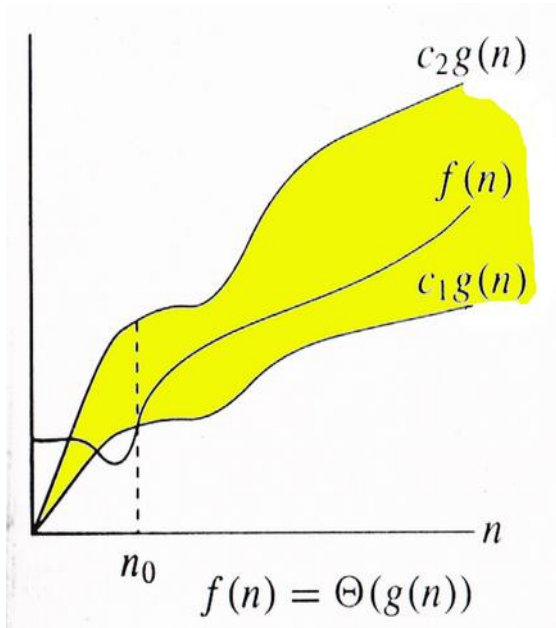
$$f(n) = O(g(n))$$

# $\Omega$ Notation

- Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm
- Let  $g$  and  $f$  be the function from the set of natural numbers to itself. The function  $f$  is said to be  $\Omega(g)$ , if there is a constant  $c > 0$  and a natural number  $n_0$  such that  $c \cdot g(n) \leq f(n)$  for all  $n \geq n_0$



# Relations Between $\Theta$ , $O$ , $\Omega$



# Properties of Asymptotic Notations

## 1. General Properties:

- If  $f(n)$  is  $O(g(n))$  then  $a \cdot f(n)$  is also  $O(g(n))$ , where  $a$  is a constant.
  - $f(n) = 2n^2 + 5$  is  $O(n^2)$
  - then,  $7 \cdot f(n) = 7(2n^2 + 5) = 14n^2 + 35$  is also  $O(n^2)$ .
  - Similarly, this property satisfies both  $\Theta$  and  $\Omega$  notation.

# Properties of Asymptotic Notations

## 1. General Properties:

- If  $f(n)$  is  $O(g(n))$  then  $a \cdot f(n)$  is also  $O(g(n))$ , where  $a$  is a constant.
  - $f(n) = 2n^2 + 5$  is  $O(n^2)$
  - then,  $7 \cdot f(n) = 7(2n^2 + 5) = 14n^2 + 35$  is also  $O(n^2)$ .
  - Similarly, this property satisfies both  $\Theta$  and  $\Omega$  notation.

# Properties of Asymptotic Notations

## 2. Transitive Properties:

If  $f(n)$  is  $O(g(n))$  and  $g(n)$  is  $O(h(n))$  then  $f(n) = O(h(n))$ .

Example:

If  $f(n) = n$ ,  $g(n) = n^2$  and  $h(n) = n^3$

$n$  is  $O(n^2)$  and  $n^2$  is  $O(n^3)$  then,  $n$  is  $O(n^3)$

Similarly, this property satisfies both  $\Theta$  and  $\Omega$  notation.

# Properties of Asymptotic Notations

## 3. Reflexive Properties:

Reflexive properties are always easy to understand after transitive.

If  $f(n)$  is given then  $f(n)$  is  $O(f(n))$ . Since MAXIMUM VALUE OF  $f(n)$  will be  $f(n)$  ITSELF!

Hence  $x = f(n)$  and  $y = O(f(n))$  tie themselves in reflexive relation always.

Example:

$$f(n) = n^2 ; O(n^2) \text{ i.e } O(f(n))$$

Similarly, this property satisfies both  $\Theta$  and  $\Omega$  notation.



# Properties of Asymptotic Notations

## 4. Symmetric Properties:

If  $f(n)$  is  $\Theta(g(n))$  then  $g(n)$  is  $\Theta(f(n))$ .

Example:

If  $f(n) = n^2$  and  $g(n) = n^2$

then,  $f(n) = \Theta(n^2)$  and  $g(n) = \Theta(n^2)$

This property only satisfies for  $\Theta$  notation.

# Properties of Asymptotic Notations

## 5. Transpose Symmetric Properties:

If  $f(n)$  is  $O(g(n))$  then  $g(n)$  is  $\Omega(f(n))$ .

Example:

If  $f(n) = n$  ,  $g(n) = n^2$

then  $n$  is  $O(n^2)$  and  $n^2$  is  $\Omega(n)$

This property only satisfies  $O$  and  $\Omega$  notations.

# Analysis of Loops

An analysis of iterative programs with simple examples is discussed.

1)  $O(1)$ : Time complexity of a function (or set of statements) is considered as  $O(1)$  if it doesn't contain loop, recursion, and call to any other non-constant time function.

For example, `swap()` function has  $O(1)$  time complexity.

# Analysis of Loops

$O(n)$ : Time Complexity of a loop is considered as  $O(n)$  if the loop variables are incremented/decremented by a constant amount. For example following functions have  $O(n)$  time complexity.

```
// Here c is a positive integer constant
```

```
for (int i = 1; i <= n; i += c) {  
    // some  $O(1)$  expressions  
}
```

```
for (int i = n; i > 0; i -= c) {  
    // some  $O(1)$  expressions  
}
```

# Analysis of Loops

$O(n^c)$ : Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example, the following sample loops have  $O(n^2)$  time complexity

```
for (int i = 1; i <= n; i += c) {  
    for (int j = 1; j <= n; j += c) {  
        // some  $O(1)$  expressions  
    }  
}
```

```
for (int i = n; i > 0; i -= c) {  
    for (int j = i+1; j <= n; j += c) {
```

# Analysis of Loops

$O(\log n)$  Time Complexity of a loop is considered as  $O(\log n)$  if the loop variables are divided/multiplied by a constant amount. And also for recursive call in recursive function the Time Complexity is considered as  $O(\log n)$ .

```
for (int i = 1; i <= n; i *= c) {  
    // some  $O(1)$  expressions  
}
```

```
for (int i = n; i > 0; i /= c) {  
    // some  $O(1)$  expressions  
}
```

# Analysis of Loops

$O(\text{LogLog}n)$  Time Complexity of a loop is considered as  $O(\text{LogLog}n)$  if the loop variables are reduced/increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
```

```
for (int i = 2; i <=n; i = pow(i, c)) {
```

```
    // some  $O(1)$  expressions
```

```
}
```

```
//Here fun is sqrt or cuberoot or any other constant root
```

```
for (int i = n; i > 1; i = fun(i)) {
```

```
    // some  $O(1)$  expressions
```

```
}
```

Let us briefly justify this intuition by using the formal definition to show that

$$\frac{1}{2}n^2 - 3n = \theta(n^2)$$

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

for all  $n \geq n_0$ . Dividing by  $n^2$  yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$

The right-hand inequality can be made to hold for any value of  $n \geq 1$  by choosing  $c_2 \geq 1/2$ . Likewise, the left-hand inequality can be made to hold for any value of  $n \geq 7$  by choosing  $c_1 \leq 1/14$ . Thus, by choosing  $c_1 = 1/14$ ,  $c_2 = 1/2$ , and  $n_0 = 7$ , we can verify that  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ . Certainly, other choices for the



### Example 1

$$\begin{aligned}f(n) &= 3\log n + 100 \\g(n) &= \log n\end{aligned}$$

Is  $f(n)$   $O(g(n))$ ? Is  $3 \log n + 100$   $O(\log n)$ ? Let's look to the definition of Big-O.

$$3\log n + 100 \leq c * \log n$$

Is there some pair of constants  $c, n_0$  that satisfies this for all  $n > n_0$ ?

$$3\log n + 100 \leq 150 * \log n, n > 2 \text{ (undefined at } n = 1)$$

Yes! The definition of Big-O has been met therefore  $f(n)$  is  $O(g(n))$ .

### *Example 2*

$$\begin{aligned}f(n) &= 3n^2 \\ g(n) &= n\end{aligned}$$

Is  $f(n) = O(g(n))$ ? Is  $3 * n^2 = O(n)$ ? Let's look at the definition of Big-O.

$$3 * n^2 \leq c * n$$

## o-notation

- The asymptotic upper bound provided by  $O$ -notation may or may not be asymptotically tight. The bound  $2n^2 = O(n^2)$  is asymptotically tight, but the bound  $2n = O(n^2)$  is not.
- We use  $o$ -notation to denote an upper bound that is not asymptotically tight.
- The main difference is that in  $f(n) = O(g(n))$ , the bound  $0 \leq f(n) \leq cg(n)$  holds for some constant  $c > 0$ , but in  $f(n) = o(g(n))$ , the bound  $0 \leq f(n) < cg(n)$  holds for all constants  $c > 0$ .
- For example,  $2n = o(n^2)$ , but  $2n^2 \neq o(n^2)$

## $\omega$ -notation

- We use  $\omega$ -notation to denote a lower bound that is not asymptotically tight.
- For example,  $n^2 / 2 = \omega(n)$ ,
- but  $n^2 / 2 \neq \omega(n^2)$

# Tractability

Some Problems are *tractable*

Some problems are *intractable*:

as they grow large, we are unable to solve them in reasonable time

What constitutes reasonable time?

- » Standard working definition: *polynomial time*
- » On an input of size  $n$  the worst-case running time is  $O(n^k)$  for some constant  $k$
- »  $O(n^2)$ ,  $O(n^3)$ ,  $O(1)$ ,  $O(n \lg n)$ ,  $O(2^n)$ ,  $O(n^n)$ ,  $O(n!)$
- » Polynomial time:  $O(n^2)$ ,  $O(n^3)$ ,  $O(1)$ ,  $O(n \lg n)$
- » Not in polynomial time:  $O(2^n)$ ,  $O(n^n)$ ,  $O(n!)$

# Polynomial-Time Algorithms

Are some problems solvable in polynomial time?

- » Of course: many algorithms we've studied provide polynomial-time solutions to some problems

Are all problems solvable in polynomial time?

- » No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given

Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

# Optimization/Decision Problems

## Optimization Problems

- » An optimization problem is one which asks, “What is the optimal solution to problem X?”
- » Examples:
  - 0-1 Knapsack
  - Fractional Knapsack
  - Minimum Spanning Tree

## Decision Problems

- » An decision problem is one with yes/no answer
- » Examples:
  - Does a graph  $G$  have a MST of weight  $\leq W$ ?

# Optimization/Decision Problems

An **optimization problem** tries to find an optimal solution

A **decision problem** tries to answer a yes/no question

Many problems will have decision and optimization versions

» Eg: Traveling salesman problem

optimization: find hamiltonian cycle of minimum weight

decision: is there a hamiltonian cycle of weight  $\leq k$

Some problems are decidable, but *intractable*:  
as they grow large, we are unable to solve them in  
reasonable time

» *Is there a polynomial-time algorithm that solves the problem?*



The class **P** consists of all problems that can be solved in polynomial time,  $O(N^k)$ , by **deterministic computers** (the ones that you have used all your life!). Examples:

- Adding two numbers:

$O(1)$  “constant”

- Extracting the element with the highest priority from a heap:

$O(\log N)$  “logarithmic” (thus,  $O(N)$  because  $N \geq \log N$  )

- Looking for an element in an array:

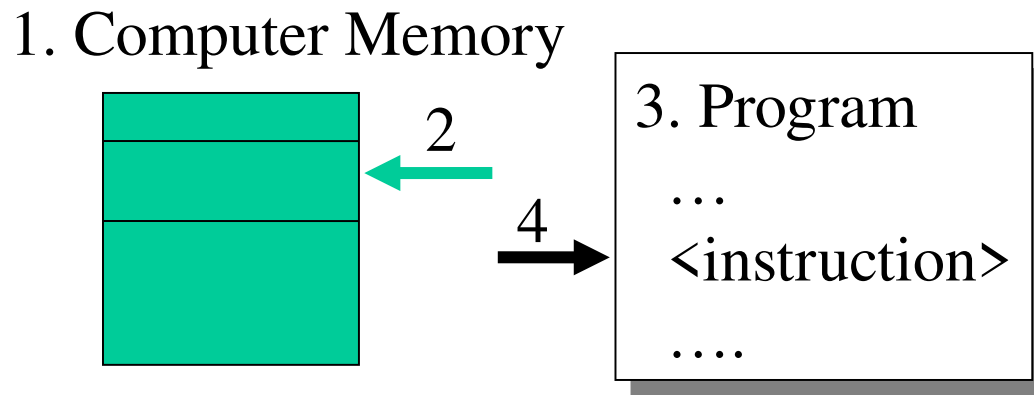
$O(N)$  “linear”

- Looking for the MST:

$O(N \log N)$  (thus,  $O(N^2)$  )

# What does Deterministic Computer Means? (Idea)

At every computational cycle we have the so-called **state of the computation**:

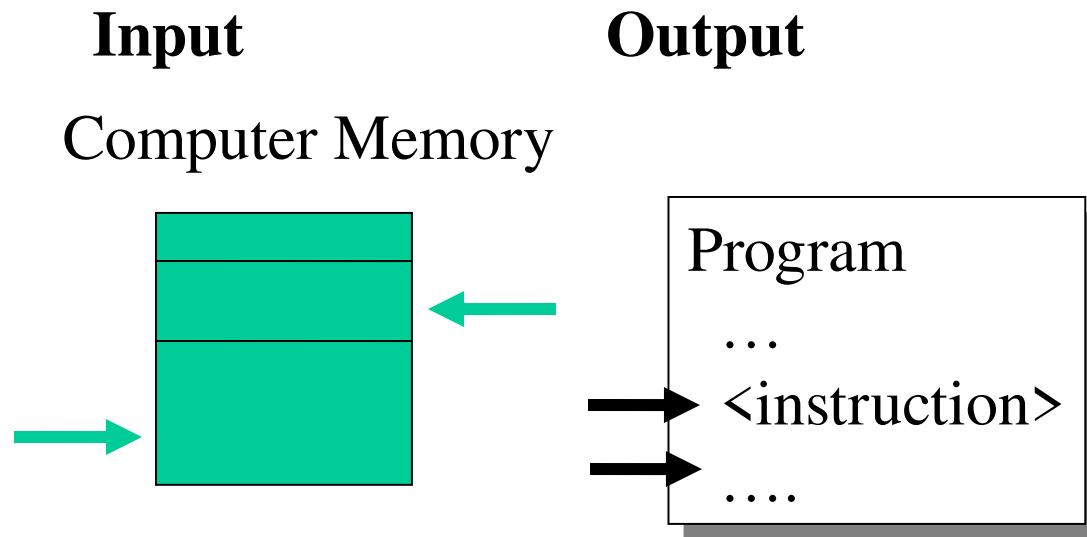


State = (1. memory, 2. location of memory being pointed at,  
3. program, 4. current instruction)

# What does Deterministic Computer Means?

## (Idea- II)

In a deterministic computer we can determine in advance for every computational cycle, the output state by looking at the input state




# Our Favorite Example

//input: an array  $A[1..N]$  and an element  $el$  that is in the array

//output: the position of  $el$  in  $A$

```
search(el, A, i)
{
    if ( $A[i] = el$ ) then return  $i$ 
    else
        return search(el, A,  $i+1$ )
}
```

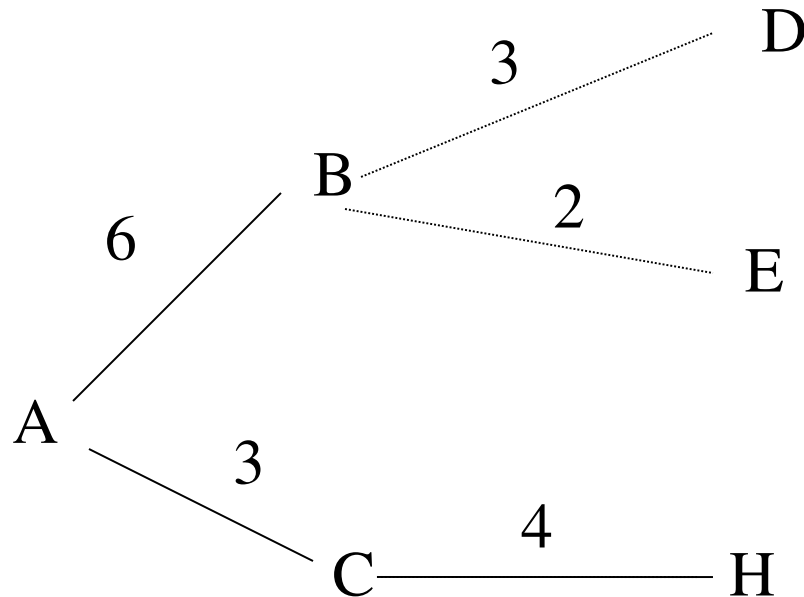
$el = 9$



7 5 3 8 9 3

Complexity:  $O(N)$

# Dijkstra's Shortest Path Algorithm

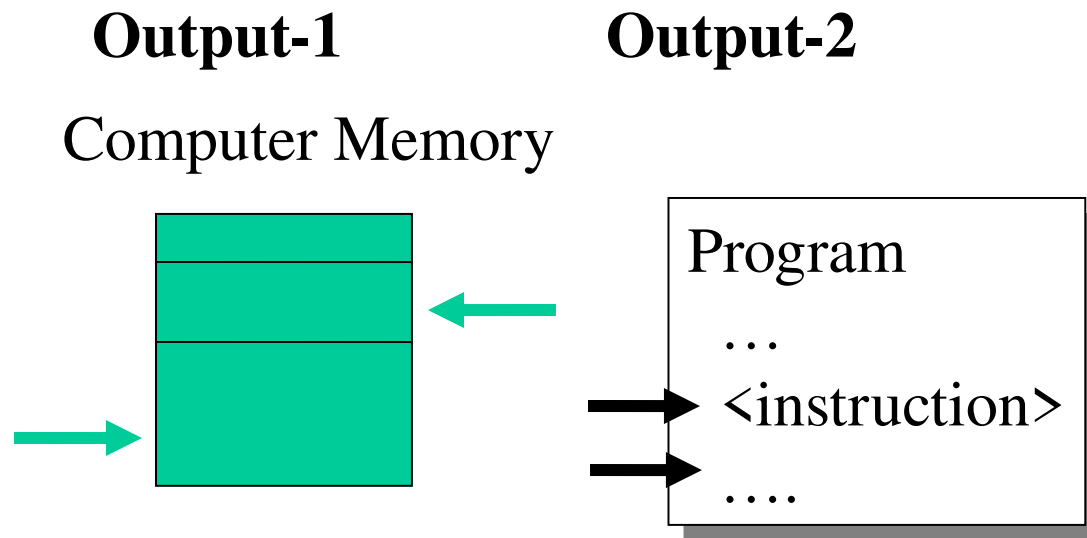


If the source is A, which edge is selected in the next iteration?

Complexity:  $O(N \log N)$  (  $N$  = number of edges + vertices)

# What does NonDeterministic Computer Means?

In a nondeterministic computer we may have more than one output state.



The computer “chooses” the correct one (“nondeterministic choice”)

# Nondeterministic Algorithm

To specify this algorithm new function was introduced:-

Choice(s)

Success()

Failure()

**Example 11.1** Consider the problem of searching for an element  $x$  in a given set of elements  $A[1 : n]$ ,  $n \geq 1$ . We are required to determine an index  $j$  such that  $A[j] = x$  or  $j = 0$  if  $x$  is not in  $A$ . A nondeterministic algorithm for this is Algorithm 11.1.

---

```
1   $j := \text{Choice}(1, n);$   
2  if  $A[j] = x$  then {write ( $j$ ); Success();}  
3  write ( $0$ ); Failure();
```

---

**Algorithm 11.1** Nondeterministic search

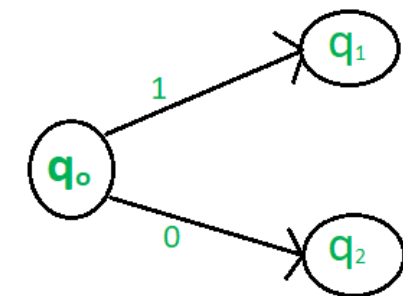


## Deterministic Algorithm

For a particular input the computer will give always same output.

Can solve the problem in polynomial time.

Can determine the next step of execution.



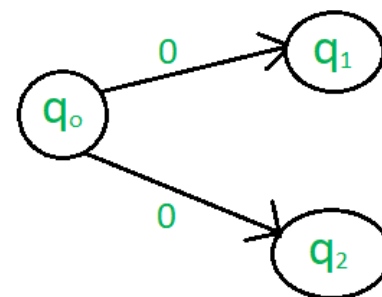
Deterministic Algorithm

## Non-deterministic Algorithm

For a particular input the computer will give different output on different execution.

Can't solve the problem in polynomial time.

Cannot determine the next step of execution due to more than one path the algorithm can take.



Non-Deterministic Algorithm

GeekforGeeks

# The Class $P$

$\underline{P}$ : the class of decision problems that have polynomial-time deterministic algorithms.

- » That is, they are solvable in  $O(p(n))$ , where  $p(n)$  is a polynomial on  $n$
- » A deterministic algorithm is (essentially) one that always computes the correct answer

# Sample Problems in P

Fractional Knapsack

MST

Sorting

Others?

# The class $NP$

$NP$ : the class of decision problems that are solvable in polynomial time on a *nondeterministic* machine (or with a nondeterministic algorithm)

(A *deterministic* computer is what we know)

A *nondeterministic* computer is one that can “guess” the right answer or solution

- » Think of a nondeterministic computer as a parallel machine that can freely release *an infinite number* of processes

Thus  $NP$  can also be thought of as the class of problems

- » whose solutions can be verified in polynomial time

Note that  $NP$  stands for “Nondeterministic Polynomial-time”

# Sample Problems in NP

Fractional Knapsack

MST

Sorting

Others?

- » Hamiltonian Cycle (Traveling Salesman)

- » Graph Coloring

- » Satisfiability (SAT)

  - the problem of deciding whether a given  
Boolean formula is satisfiable

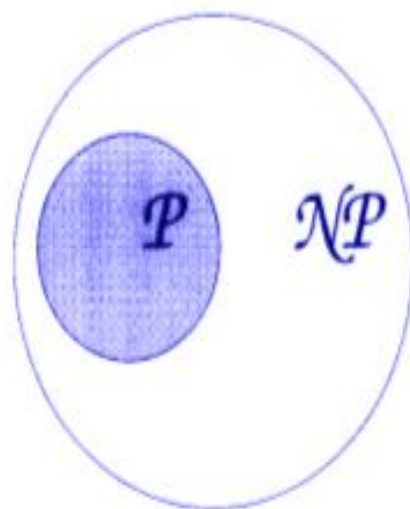


Figure 11.1 Commonly believed relationship between  $\mathcal{P}$  and  $\mathcal{NP}$

## NP Problem:

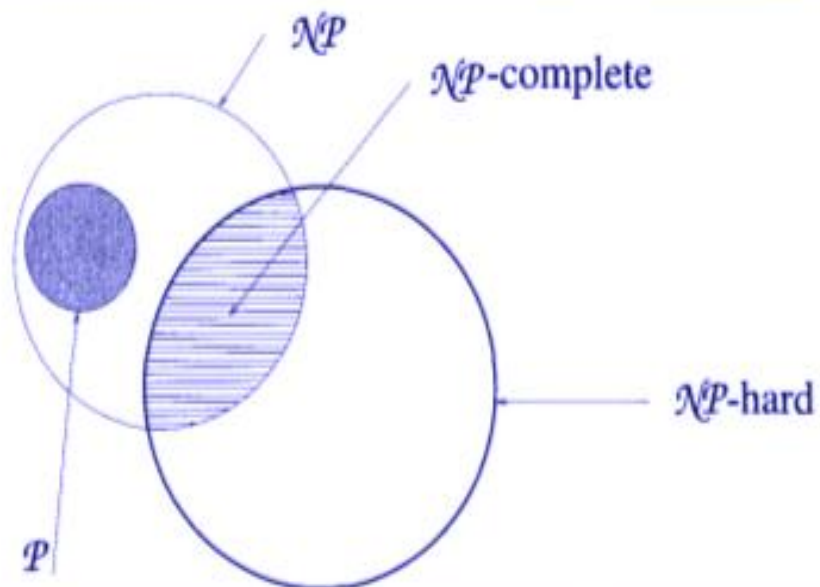
Set of problems whose solutions are hard to find but easy to verify and are solved by Non-Deterministic Machine in polynomial time.

## NP-Complete Problem:

A problem  $X$  is NP-Complete if there is an NP problem  $Y$ , such that  $Y$  is reducible to  $X$  in polynomial time. NP-Complete problems are as hard as NP problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic Turing machine can solve NP-Complete problem in polynomial time.

## NP-Hard Problem:

A Problem  $X$  is NP-Hard if there is an NP-Complete problem  $Y$ , such that  $Y$  is reducible to  $X$  in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.



**Figure 11.2** Commonly believed relationship among  $\mathcal{P}$ ,  $\mathcal{NP}$ ,  $\mathcal{NP}$ -complete, and  $\mathcal{NP}$ -hard problems



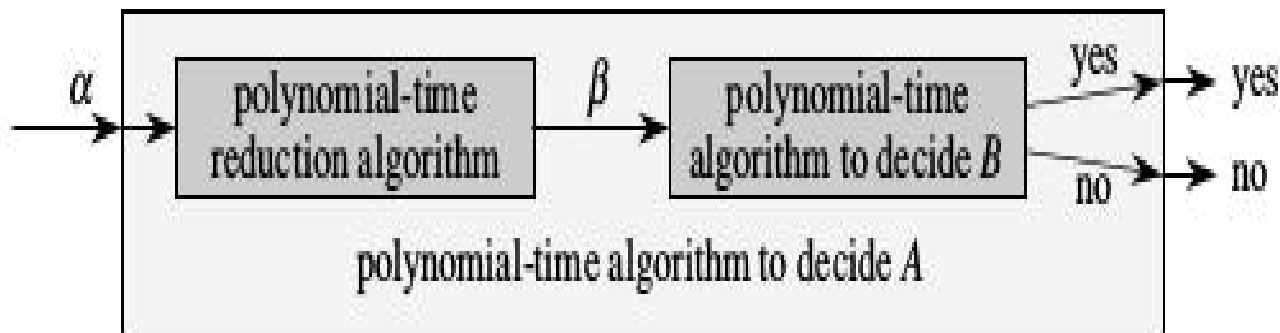
# NP-completeness and Reducibility

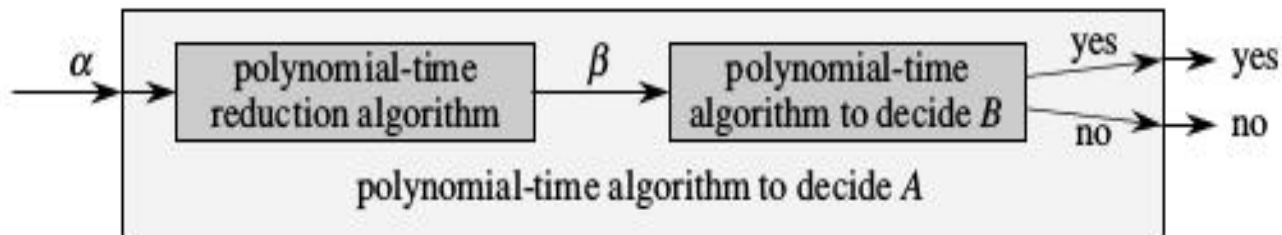
A (class of) problem  $P_1$  is **poly-time reducible** to  $P_2$ , written as  $P_1 \leq_p P_2$  if there exists a poly-time function  $f$ :

$P_1 \rightarrow P_2$  such that for any instance of  $p_1 \in P_1$ ,  $p_1$  has “YES” answer if and only if answer to  $f(p_1)$  ( $\in P_2$ ) is also “YES”.

*Theorem 34.3:*

For all problems  $A$  and  $B$ , if  $A \leq_p B$  and  $B \in P$ , then  $A \in P$ .





We call such a procedure a polynomial-time reduction algorithm and, as it provides us a way to solve problem A in polynomial time:

1. Given an instance  $\alpha$  of problem A, use a polynomial-time reduction algorithm to transform it to an instance  $\beta$  of problem B.
2. Run the polynomial-time decision algorithm for B on the instance  $\beta$ .
3. Use the answer for  $\beta$  as the answer for  $\alpha$ .

# Polynomial-time Verification

Suppose that for a given instance  $(G, u, v, k)$  of the decision problem PATH, we are also given a path  $p$  from  $u$  to  $v$ . We can easily check whether the length of  $p$  is at most  $k$ , and if so, we can view  $p$  as a “certificate” that the instance indeed belongs to PATH.

For the decision problem PATH, this certificate doesn’t seem to buy us much. After all, PATH belongs to P—in fact, PATH can be solved in linear time—and so verifying membership from a given certificate takes as long as solving the problem from scratch.

# Hamiltonian cycles

Formally, a hamiltonian cycle of an undirected graph  $G = (V, E)$  is a simple cycle that contains each vertex in  $V$ .

A graph that contains a hamiltonian cycle is said to be hamiltonian; otherwise, it is nonhamiltonian.

We can define the hamiltonian-cycle problem, “Does a graph  $G$  have a hamiltonian cycle?”

as a formal language:

$\text{HAM-CYCLE} = \{(G) : G \text{ is a hamiltonian graph}\}$ .

# Hamiltonian cycles: Verification

Time complexity:-  $O(N * N!)$

Given graph  $G$  is hamiltonian, and then prove it by giving you the vertices in order along the hamiltonian cycle.

It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of  $V$  and whether each of the consecutive edges along the cycle actually exists in the graph.

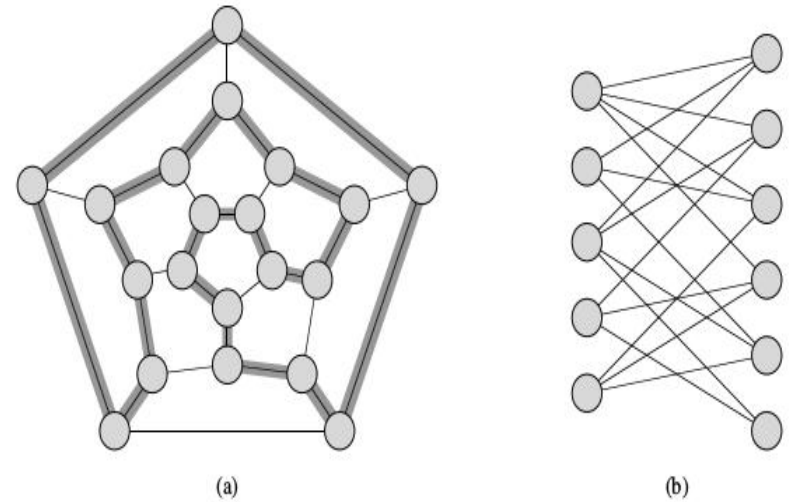


Figure 34.2 (a) A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges. (b) A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

# Hamiltonian cycles: Verification

We define a verification algorithm as being a two-argument algorithm  $A$ , where one argument is an ordinary input string  $x$  and the other is a binary string  $y$  called a certificate.

A two-argument algorithm  $A$  verifies an input string  $x$  if there exists a certificate  $y$  such that  $A(x, y) = 1$ . The language verified by a verification algorithm  $A$  is

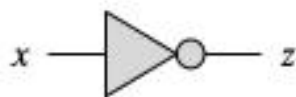
$$L = \{x \in \{0, 1\}^* : \text{there exists } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\} .$$

# Circuit satisfiability

A boolean combinational element is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function.

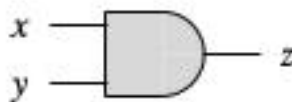
Boolean values are drawn from the set  $\{0, 1\}$ , where 0 represents FALSE and 1 represents TRUE .

Basic logic gates that we use in the circuit-satisfiability problem: the NOT gate (or inverter), the AND gate, and the OR gate.



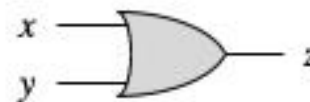
$x$	$\neg x$
0	1
1	0

(a)



$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(b)

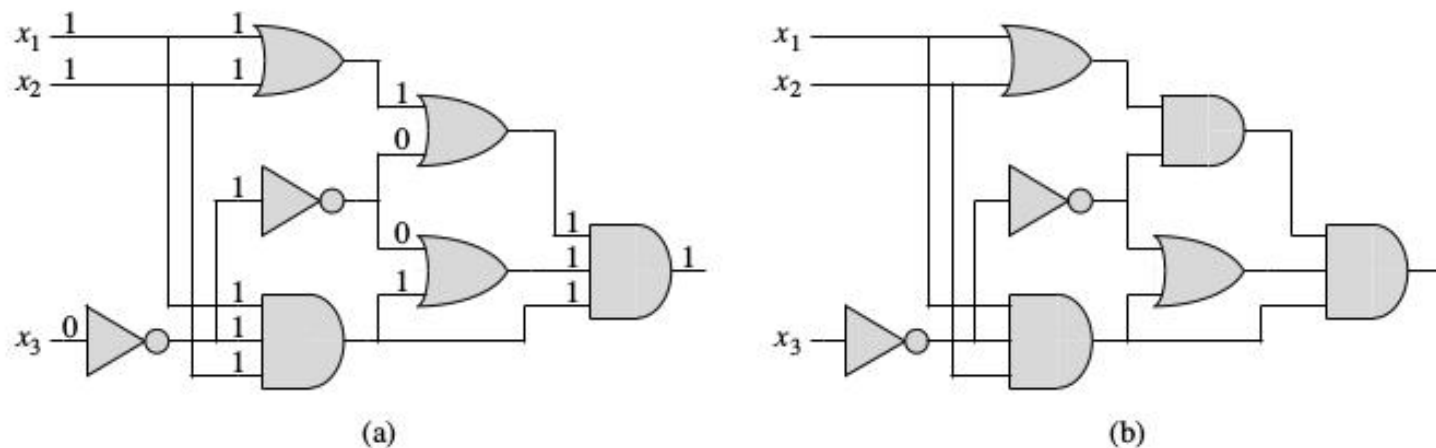


$x$	$y$	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

(c)

# Circuit satisfiability

A boolean combinational circuit consists of one or more boolean combinational elements interconnected by wires. A wire can connect the output of one element to the input of another, thereby providing the output value of the first element as an input value of the second.



**Figure 34.8** Two instances of the circuit-satisfiability problem. (a) The assignment  $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$  to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. (b) No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.



# Circuit satisfiability

The circuit-satisfiability problem is, “Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?”

In order to pose this question formally, however, we must agree on a standard encoding for circuits.

The size of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit.

One can devise a graphlike encoding that maps any given circuit  $C$  into a binary string  $(C)$  whose length is polynomial in the size of the circuit itself.

As a formal language, we can therefore define

$\text{CIRCUIT-SAT} = \{(C) : C \text{ is a satisfiable boolean combinational circuit}\} .$

Base Problem to relate  
exponential type  
Problem

# Satisfiability Problem

3-SAT Problem is the  
problem of determining  
the satisfiability of a  
formula in conjunctive  
normal form (CNF) where  
each clause is limited to at  
most 3 literals

Conjunctive Normal Form(CNF)-Satisfiability formula using

Boolean variables( $x_1, x_2, x_3$ )

Ex:

$$(x_1 \cup x_2 \cup \neg x_3) \cap (\neg x_1 \cup x_2 \cup \neg x_3)$$

Satisfiability problem is to  
find out for what values  
this formula is true

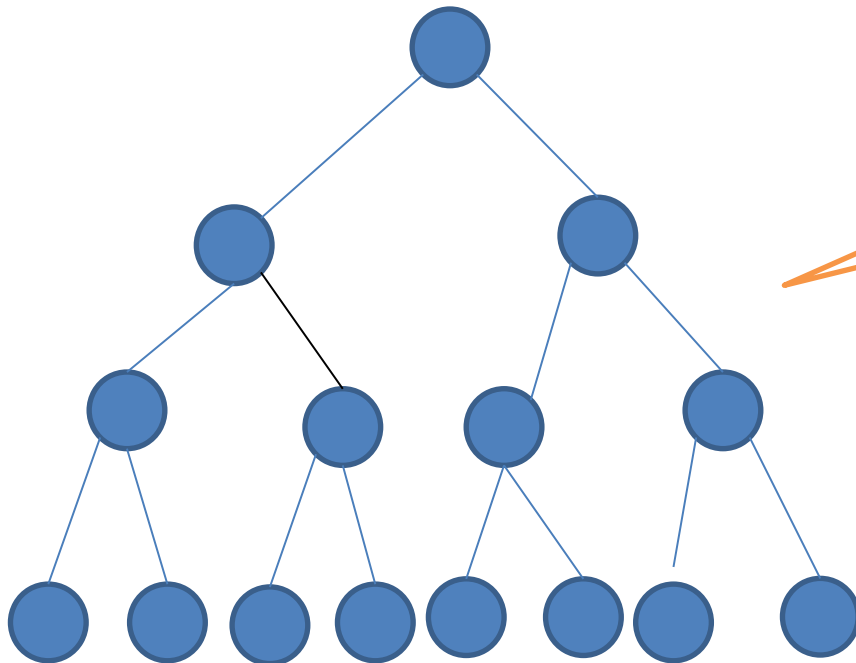
$2^3$  for n values  
 $2^n$

x1	x2	x3
0	0	0
0	0	1
0	1	0
1	0	0
1	1	0
1	0	1
0	1	1
1	1	1

# How to relate

If SAT problem is reduced to another exponential problem in polynomial time then solution of SAT can be used to solve reduced problem and vice versa

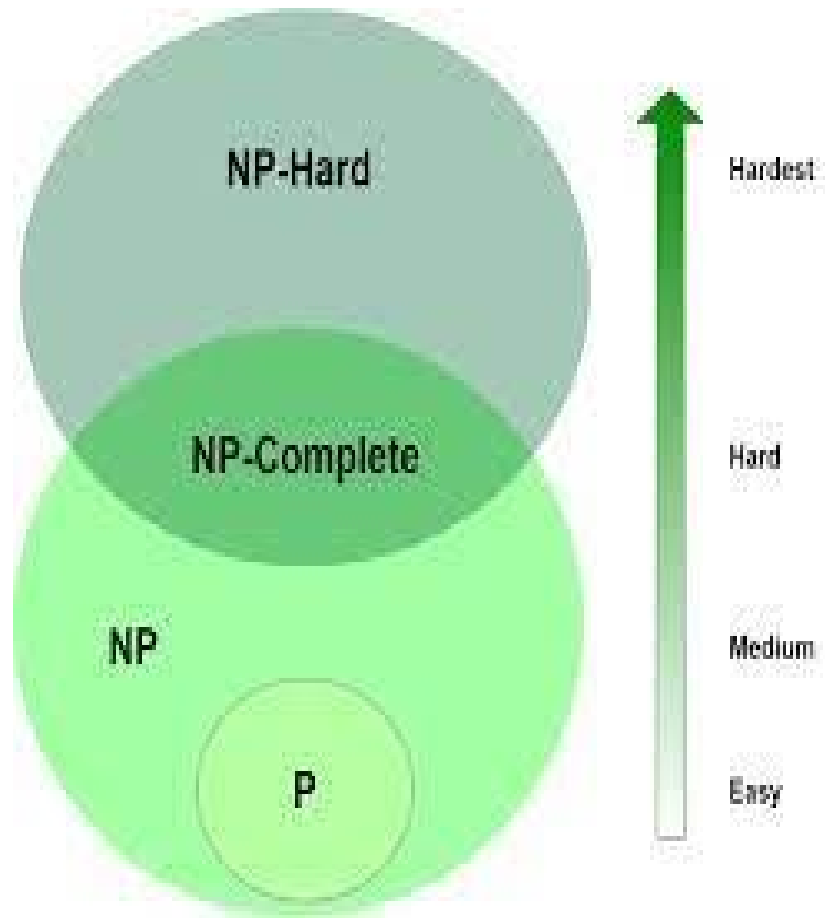
EX: Draw state space tree to solve SAT problem



0/1 knapsack problem also can be solved using this Ex.  $n=3$   $M=8$   
 $w=\{5,4,3\}$   $p=\{10,8,12\}$

So, if we are able to solve this SAT problem in polynomial time we can solve 0/1 knapsack in polynomial time. This way we can show relationship by taking SAT as base problem

# NP-Hard, NP-Complete, NP, P Class Problems



SAT problems are NP hard problems and the problems, which are reduction of SAT problems are also NP-Hard problem.

SAT problems are NP-Hard as well as NP complete(Non deterministic polynomial algorithm is available (proved))

# Non Deterministic Clique Problem

## Clique Problem:

In an undirected graph, a **clique** is a complete sub-graph of the given graph. Complete sub-graph means, all the vertices of this sub-graph is connected to all other vertices of this sub-graph.

The **Max-Clique** problem is the computational problem of finding maximum clique of the graph. Max clique is used in many real-world problems.

Ex: Let us consider a social networking application, where vertices represent people's profile and the edges represent mutual acquaintance in a graph. In this graph, a clique represents a subset of people who all know each other

# Non Deterministic Clique Problem cont..

To find a maximum clique, one can systematically inspect all subsets, but this sort of brute-force search is too time-consuming for networks comprising more than a few dozen vertices.

## Algorithm: Max-Clique ( $G, n, k$ )

$S := \Phi$

for  $i = 1$  to  $k$  do

$t := \text{choice}(1 \dots n)$

    if  $t \in S$  then

        return failure

$S := S \cup t$

for all pairs  $(i, j)$  such that  $i \in S$  and  $j \in S$  and  $i \neq j$

do

    if  $(i, j)$  is not a edge of the graph then

        return failure

return success

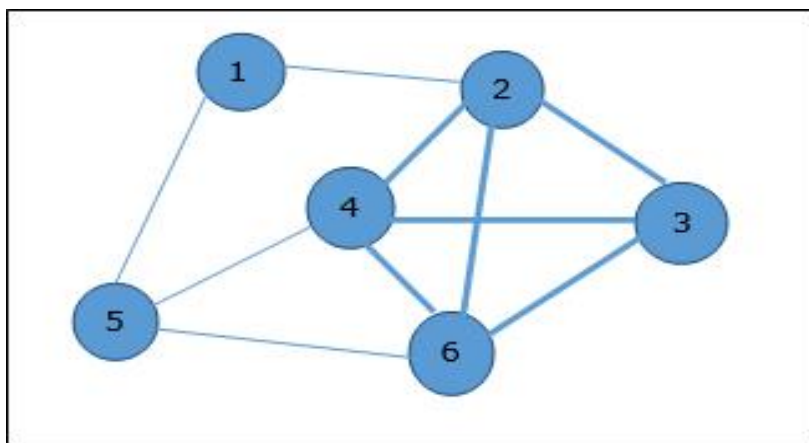
### Analysis

Max-Clique problem is a non-deterministic algorithm. In this algorithm, first we try to determine a set of  $k$  distinct vertices and then we try to test whether these vertices form a complete graph.

There is **no polynomial time deterministic algorithm** to solve this problem. This problem is NP-Complete.

# Non Deterministic Clique Problem example

Take a look at the graph. Here, the sub-graph containing vertices 2, 3, 4 and 6 forms a complete graph. Hence, this sub-graph is a **clique**. As this is the maximum complete sub-graph of the provided graph, it's a **4-Clique**.

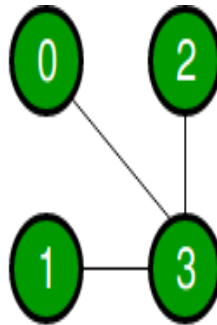


# Vertex Cover Problem

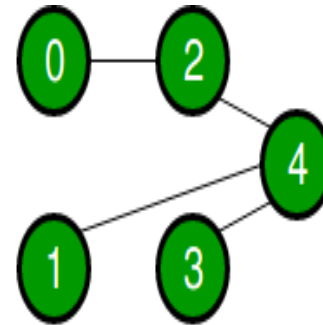
- A vertex cover of an undirected graph is a subset of its vertices such that for every edge  $(u, v)$  of the graph, either 'u' or 'v' is in the vertex cover.
- Although the name is Vertex Cover, the set covers all edges of the given graph.
- *Given an undirected graph, the vertex cover problem is to find minimum size vertex cover*



Minimum vertex cover is empty{}



Minimum vertex cover is {3}



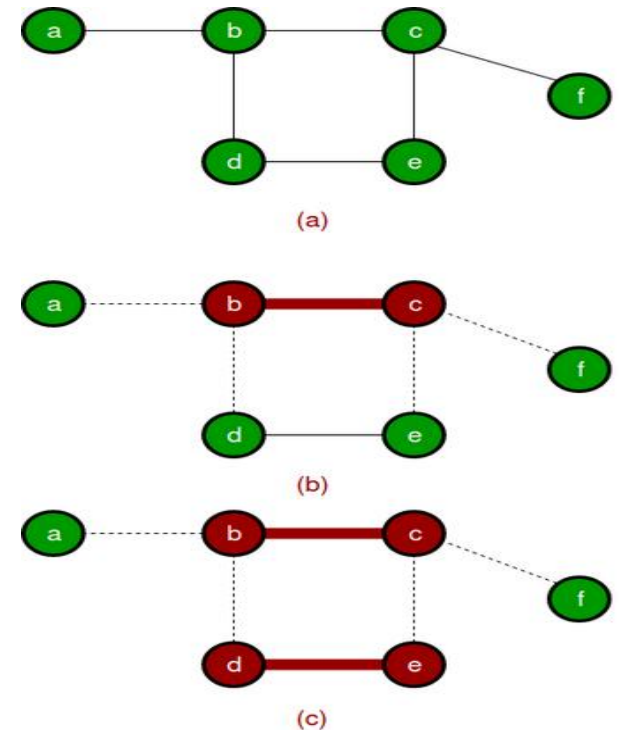
Minimum vertex cover is {4, 2} or {4, 0}



# Approximate Algorithm for Vertex Cover:

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time.

- 1) Initialize the result as  $\{\}$
- 2) Consider a set of all edges in given graph. Let the set be E.
- 3) Do following while E is not empty
  - ...a) Pick an arbitrary edge (u, v)
  - from set E and add 'u' and 'v' to result
  - ...b) Remove all edges from E which are either incident on u or v.
- 4) Return result



Minimum Vertex Cover is {b, c, d} or {b, c, e}

# Proof that vertex cover is NP complete

**Problem** – Given a graph  $G(V, E)$  and a positive integer  $k$ , the problem is to find whether there is a subset  $V'$  of vertices of size at most  $k$ , such that every edge in the graph is connected to some vertex in  $V'$ .

## Explanation –

First let us understand the notion of an instance of a problem.

- ✓ An instance of a problem is nothing but an input to the given problem.
- ✓ An instance of the Vertex Cover problem is a graph  $G(V, E)$  and a positive integer  $k$ , and the problem is to check whether a vertex cover of size at most  $k$  exists in  $G$ .

# **Proof that vertex cover is NP complete cont..**

Since an NP Complete problem, by definition, is a problem which is both in NP and NP hard, the proof for the statement that a problem is NP Complete consists of two parts:

- 1. Proof that vertex cover is in NP**
- 2. Proof that vertex cover is NP Hard**

# Proof that vertex cover is in NP

If any problem is in NP, then, given a ‘certificate’ (a solution) to the problem and an instance of the problem (a graph  $G$  and a positive integer  $k$ , in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time.

The certificate for the vertex cover problem is a subset  $V'$  of  $V$ , which contains the vertices in the vertex cover.

# Proof that vertex cover is in NP cont..

We can check whether the set  $V'$  is a vertex cover of size  $k$  using the following strategy (for a graph  $G(V, E)$ ):

```
let count be an integer
set count to 0
for each vertex  $v$  in  $V'$ 
    remove all edges adjacent to  $v$  from
set  $E$ 
    increment count by 1
if count =  $k$  and  $E$  is empty
    then
        the given solution is correct
else
    the given solution is wrong
```

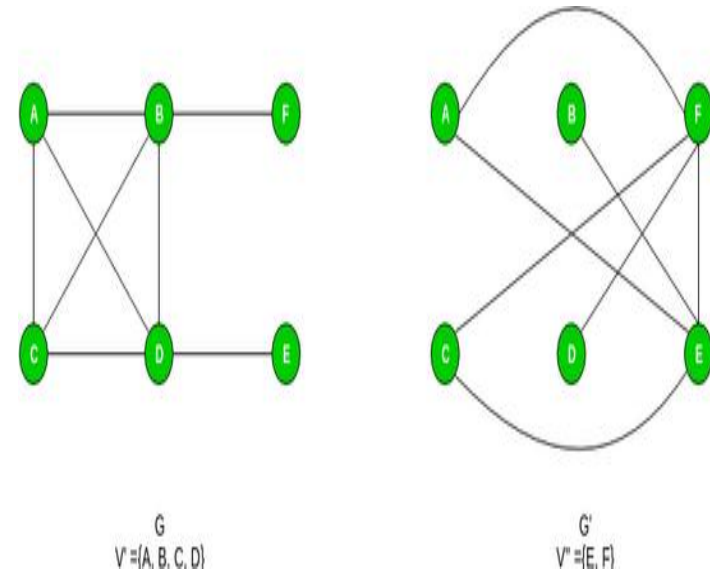
It is plain to see that this can be done in polynomial time. Thus the vertex cover problem is in the class NP.

## Proof that vertex cover is NP Hard cont..

- ✓ Here, we consider the problem of finding out whether there is a clique of size  $k$  in the given graph.
- ✓ Therefore, an instance of the clique problem is a graph  $G (V, E)$  and a non-negative integer  $k$ , and we need to check for the existence of a clique of size  $k$  in  $G$ .

# Proof that vertex cover is NP Hard cont..

- ✓ Now, we need to show that any instance  $(G, k)$  of the Clique problem can be reduced to an instance of the vertex cover problem.
- ✓ Consider the graph  $G'$  which consists of all edges not in  $G$ , but in the complete graph using all vertices in  $G$ . Let us call this the complement of  $G$ .



Now, the problem of finding whether a clique of size  $k$  exists in the graph  $G$  is the same as the problem of finding whether there is a vertex cover of size  $|V| - k$  in  $G'$ .

We need to show that this is indeed the case.

# Proof that vertex cover is NP Hard cont..

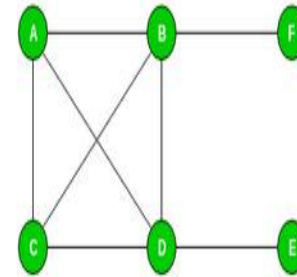
- ✓ Assume that there is a clique of size  $k$  in  $G$ .
- ✓ Let the set of vertices in the clique be  $V'$ . This means  $|V'| = k$ .
- ✓ In the complement graph  $G'$ , let us pick any edge  $(u, v)$ .
- ✓ Then at least one of  $u$  or  $v$  must be in the set  $V - V'$ .
- ✓ This is because, if both  $u$  and  $v$  were from the set  $V'$ , then the edge  $(u, v)$  would belong to  $V'$ , which, in turn would mean that the edge  $(u, v)$  is in  $G$ .
- ✓ This is not possible since  $(u, v)$  is not in  $G$ . Thus, all edges in  $G'$  are covered by vertices in the set  $V - V'$ .



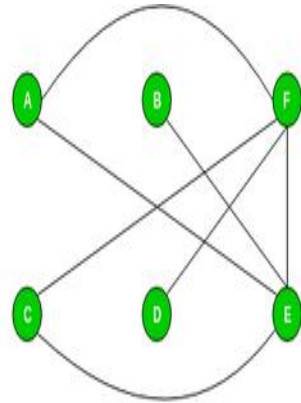
# Proof that vertex cover is NP Hard

## cont..

- ✓ Now assume that there is a vertex cover  $V''$  of size  $|V| - k$  in  $G'$ . This means that all edges in  $G'$  are connected to some vertex in  $V''$ .
- ✓ As a result, if we pick any edge  $(u, v)$  from  $G'$ , both of them cannot be outside the set  $V''$ .
- ✓ This means, all edges  $(u, v)$  such that both  $u$  and  $v$  are outside the set  $V''$  are in  $G$ , i.e., these edges constitute a clique of size  $k$ .



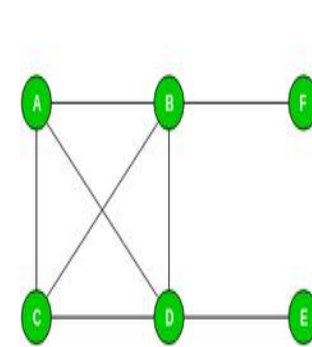
G  
 $V'' = \{A, B, C, D\}$



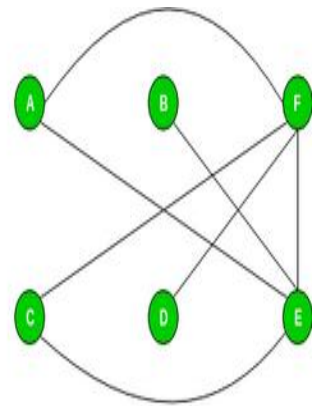
$G'$   
 $V'' = \{E, F\}$

# Proof that vertex cover is NP Hard cont..

Thus, we can say that there is a clique of size  $k$  in graph  $G$  if and only if there is a vertex cover of size  $|V| - k$  in  $G'$ , and hence, any instance of the clique problem can be reduced to an instance of the vertex cover problem.



$G$   
 $V = \{A, B, C, D, E, F\}$



$G'$   
 $V' = \{E, F\}$

Thus, vertex cover is NP Hard.  
Since vertex cover is in both NP and NP Hard classes, it is NP Complete.

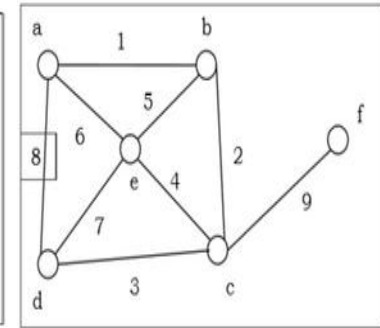
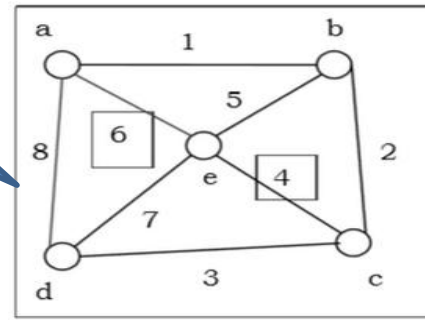
# Hamiltonian Cycle

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once.

A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path.

Hamiltonian  
cycle

# Hamiltonian Cycle



Non-  
Hamiltonian  
cycle

**Problem** Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

## **Input:**

A 2D array  $\text{graph}[V][V]$  where  $V$  is the number of vertices in graph and  $\text{graph}[V][V]$  is adjacency matrix representation of the graph. A value  $\text{graph}[i][j]$  is 1 if there is a direct edge from  $i$  to  $j$ , otherwise  $\text{graph}[i][j]$  is 0.

## **Output:**

An array  $\text{path}[V]$  that should contain the Hamiltonian Path.  $\text{path}[i]$  should represent the  $i$ th vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the

# Proof that Hamiltonian Cycle is NP-Complete

## Hamiltonian Cycle:

A cycle in an undirected graph  $G=(V, E)$  which traverses every vertex exactly once.

## Problem Statement:

Given a graph  $G(V, E)$ , the problem is to determine if the graph contains a Hamiltonian cycle consisting of all the vertices belonging to  $V$ .

# Proof that Hamiltonian Cycle is NP-Complete

## cont...

### Explanation

An instance of the problem is an input specified to the problem.

- ✓ An instance of the Independent Set problem is a graph  $G(V, E)$ , and the problem is to check whether the graph can have a Hamiltonian Cycle in  $G$ .
- ✓ Since an NP-Complete problem, by definition, is a problem which is both in NP and NP-hard, the proof for the statement that a problem is NP-Complete consists of two parts:

# Proof that Hamiltonian Cycle is NP-Complete cont...

Refer link:

<https://www.geeksforgeeks.org/proof-that-hamiltonian-cycle-is-np-complete>

- ✓ The problem itself is in NP class.
- ✓ All other problems in NP class can be polynomial-time reducible to that.

(B is polynomial-time reducible to C)

$B \leq C$

✓ If the 2nd condition is only satisfied then the problem is called NP-Hard.