UNIT-1

What is an Algorithm?

An algorithm is a finite sequence of instructions for solving a computational problem.

In addition, all algorithms must satisfy the following criteria:

1. **Input**: Zero or more quantities are externally supplied.
2. **Output:** At least one quantity is produced as output.
3. **Definiteness**: Each instruction must be clear and unambiguous.
4. **Finiteness**: The algorithm must terminate after a finite number of steps.
5. **Effectiveness:** Each instruction must be feasible, that means a person should be able to carry out the instruction correctly by hand in a finite length of time. And algorithm must not contain unnecessary or redundant instructions.

In formal computer science, one distinguishes between an algorithm and a program.
A program does not necessarily satisfy the 4^{th} condition (finiteness). One important example for such a program for a computer is its OS, which never terminates except for system crashes or when system is turned off, but continues to loop until a new job is entered.

Since our programs always terminate, we use algorithm and program interchangeably.

Pseudocode conventions for expressing Algorithms:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces: { and }.
   - Statements are delimited by ; .

3. An identifier begins with a letter.
-The data types of variables are not explicitly declared. The types will be clear from the context.
Whether a variable is global or local to a procedure will also be evident from the context.

4. Compound data types can be formed with records.
Example:

```
node = record
{
datatype_1 data1;
datatype_2 data2;
.
datatype_n  data-n;
node *link;
}
```

   - Here, link is a pointer to the record type node.

5. Assignment of values to variables is done using the assignment statement.
   <*variable*>: = <*expression*>;

6. There are two Boolean values ***true*** and ***false***.
   Logical operators: *AND*, *OR*, and *NOT*
   Relational operators: <, >, ≤, ≥, =, ≠

7. Elements of multidimensional arrays are accessed using [and].
   Example:
   If A is a 2D array, the $(i , j)^{th}$ element of the array is denoted as A[i , j].
   - Array indices start at 1.

8.The following looping statements are employed:
for, **while**, and **repeat-until**.

The **while** loop takes the following form:

while<*condition*> **do**
{
<*statement 1*>
…
<*statement n*>
}

The general form of **for** loop:

for *variable*: = *value1* **to** *value2* **step** *step_size* **do**
{
<*statement 1*>
…
<*statement n*>
}

The clause "**step** *step_size* " is optional and taken as +1 if it is not present. *step_size* could either be positive or negative.

A **repeat-until** statement is constructed as follows:

repeat
<*statement 1*>
…
<*statement n*>
until<*condition*>

The statements are executed as long as<*condition*> is false. The value of *condition* is computed after executing the statements.

The instruction **break**; can be used within any of the above looping instructions to force **exit**.

9. A conditional statement has the following forms:
if<*condition*> **then** <*statement*>

if<*condition*> **then** <*statement 1*> **else** <*statement 2*>
case statement
case
{
: <*condition 1*> : <*statement 1*>
: <*condition 2*> : <*statement 2*>
…
: <*condition n*> : <*statement n*>
: **else**: <*statement n+1*>
}

10. Input & output are done using the instructions **read** and **write**.

11. There is only one type of procedure (or, function): **Algorithm**
- An algorithm consists of a heading and a body.
- The heading takes the form:
  Algorithm Name (< parameter list >)
- The body has one or more statements enclosed within braces.
- An algorithm may or may not return any values.
- Simple variables to procedures are passed by values.
- Arrays and records are passed by reference.
  An array name or a record name is treated as a pointer to the respective datatype.

Example: Write an algorithm that finds and returns the maximum of 'n' given numbers.
Solution:

```
Algorithm Max(A, n)
// A is an array of size n.
{
    Result := A[1];
    for i := 2 to n do
        if A[i]> Result then Result := A[i];
            return Result;
}
```

In the above algorithm, 'A' and 'n' are procedure parameters. 'Result' and 'i' are local variables.

SELECTION SORT

Suppose we want to devise an algorithm that sorts a collection of n elements of arbitrary type.

A Simple solution is given by the following statement:
"*From those elements that are currently unsorted, find the smallest and place it next in the sorted list.*"

The above statement is not an algorithm because it leaves several questions unanswered. For example, it doesn't tell us "where and how the elements are initially stored, or where we should place the result."

We assume that, the elements are stored in an array 'a' such that the i^{th} element is stored in the i^{th} position, i.e., a[i] , $1 \le i \le n$.
And we also assume that the sorted elements are also stored in the same array 'a'.
Algorithm:

```
for i := 1 to n do
{
    Examine a[i] to a[n] and suppose the smallest element is at a[j];
    Interchange a[i] and a[j];
}
```

To turn the above algorithm into a pseudocode program, two clearly defined subtasks remain:
1. Finding the smallest element (say, a[j]).
2. Interchanging it with a[i].

Note: To denote the range of array elements, a[1] through a[n], we use the notation a[1: n].

```
Algorithm SelectionSort (a, n)
{
    // sort the array a[1:n] into non decreasing order.
```

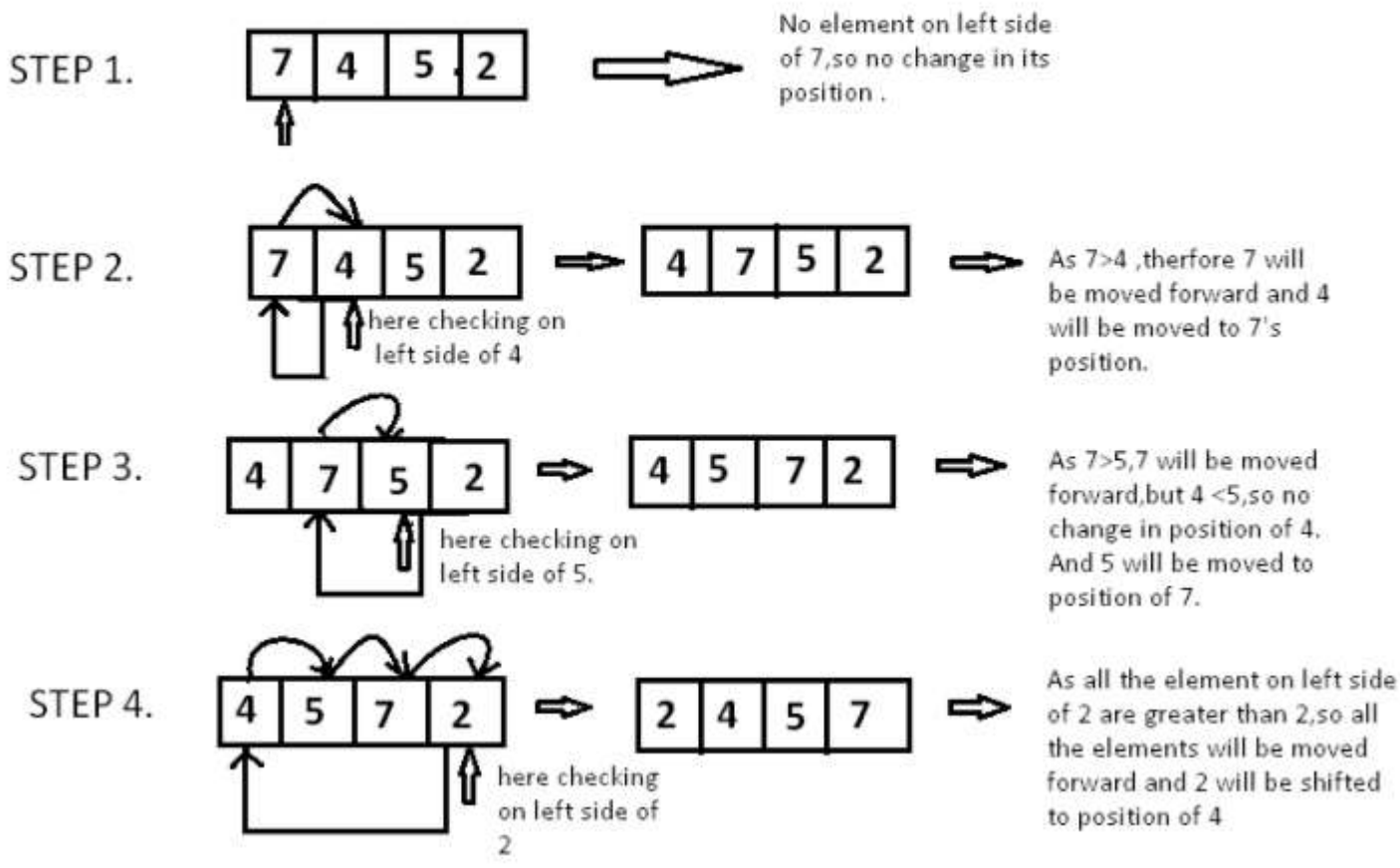
```
        for i := 1 to n-1 do
        {
                //Find out the index of the smallest element from a[i:n] and place it in j.
                j:= i;
                for k := i+1 to n do
                {
                        if (a[k] < a[j]) then
                                { j:= k; }
                }
                if (j ≠ i) then
                {
                        // interchange a[i] and a[j]
                        t := a[i];
                        a[i] := a[j];
                        a[j] := t;
                }
        }
}
```

Algorithm for Insertion Sort:

```
Algorithm InsertionSort (a, n)
{
        // sort the array a[1:n] into non decreasing order.
        for  i := 2 to n do
        {
                key=a[i];
                // Insert a[i] into the sorted part of the array, i.e., a [1: i-1]
                   j=i-1;
                while(j>=1 and a[j]>key) do
                {
                        a[j+1]=a[j];  // move a[j] to its next position  in the right side
                        j=j-1;
                }
                a[j+1]=key;

        }
}
```

Example:

Take array $A[] = [7, 4, 5, 2]$.



Performance analysis:-

To judge the performance of an algorithm we use two terms.

1)Space complexity

2)Time complexity

1)Space complexity:-

The space complexity of an algorithm is the amount of memory it needs to run to completion.

2)Time complexity:-

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Space complexity:-
The space needed by an algorithm is seen to be the sum of the following components.
→A fixed part that is independent of the characteristics of the inputs and output. This part typically includes the instruction space, space for simple variables & fixed size component variables, space for constants and so on.
→Variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by reference variables (to the extent that this depends on instance characteristics), and the recursion stack space.
→The space requirement S(P) of any algorithm P may therefore be written as,
$S(P) = C + S_P$(instance characteristics).
Where, C is a constant.
→When analysing the space complexity of an algorithm, we concentrate solely on estimating
S_P(instance characteristics).
For any given problem, first we need to determine which instance characteristics to use to measure the space requirements. Generally speaking, our choices are limited to quantities related to the number and size of the input and outputs of the algorithm.

EXAMPLE-1:-

```
Algorithm abc(a, b, c)
{
    return a+b+b*c+(a+b-c)/(a+b)+40;
}
```

→For this algorithm, the problem instance is characterised by the specific values of *a, b, c*.
Assume that one word is sufficient to store the values of each *a, b, c* and the result. So, the space needed by algorithm *abc* is 4 words.
So, the space needed by abc is independent of the instance characteristics. Consequently, S_P(instance characteristics) is equal to 0.
So, space needed by this algorithm is constant.

EXAMPLE 2: Iterative algorithm for sum of n real numbers

```
Algorithm sum (a, n)
{
 S=0.0;
for i:=1 to n do
    S:=S+a[i];
return S;
}
```

For this algorithm, the problem instance is characterised by *n* (means value of *n*).
The space needed for array address *'a'* is one word. The space needed by *n* is one word. Space needed by *i* and *S* are one word and one word.
So, we obtain $S_{sum}(n) = 0$.
So, space complexity of the algorithm *sum* is, S(sum) = constant.

EXAMPLE-3:-
Recursive algorithm for sum of n numbers:

```
Algorithm Rsum(a, n)
{
  if (n<=0) then return 0.0;
  else return Rsum(a, n-1) + a(n);
}
```

For this algorithm, the problem instance is characterised by *n*.
The recursion stack space includes space for the formal parameters, local variables, and the return address. Assume that return address requires one word of memory and one word is required for each of the formal parameters *'a'* and *'n'*. Since the depth of the recursion is *n*, the recursion stack space needed is *3n*.
So, $S_{Rsum}(n) = 3n$.
S(Rsum)=C+3n = O(n).

EXAMPLE-4:-

```
Algorithm copy(a, n)
{
      for i:=1 to n do
          b[i]:=a[i];
}
```

Array *'b'* needs *'n'* memory locations.
So, S_{copy}(instance characteristics) = $S_{copy}(n) = n$.
S(copy) = C+n = O(n).

Time complexity:-
The time T(P) taken by an algorithm P, is the sum of the compile time and run time.
→Compile time is fixed. It does not depend on the instance characteristics.
→Consequently, we concern ourselves with just the run time of an algorithm.
→The runtime of algorithm P is denoted by t_P(instance characteristics), where instance characteristics are those parameters that characterize the problem instance .

Determining the time complexity by step count method:
→The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function for which it was written.
→The no. of steps is itself a function of the instance characteristics.
Usually, we choose those characteristics that are of importance for us.
→Once the relevant instance characteristics (such as n, m, p, q, ...) have been selected, we can define what a step is.
→A step is any computational unit that is independent of the instance characteristics (n, m, p, q, ...).
EXAMPLE:-
- 10 additions can be one step.

- 100 multiplications is one step.

But

- n additions cannot be one step, or
- m/2 additions cannot be one step, or
- p+q subtractions cannot be one step.

Method to determine the step count of an algorithm:-

(1) Determine the total no. of times each statement is executed (i.e., frequency).
(2) Determine the no. of steps per execution(s/e) of the statement.
(3) Multiply the above two quantities to obtain the total no. of steps contributed by each statement.
(4) Add the contributions of all statements to obtain the step count for the entire algorithm.

EXAMPLE-1:-

Statement	No. of times of execution (frequency)	s/e	total no. of steps per statement

Algorithm sum(a,n)
{
s:=0.0;
for i:=1 to n do
s:=s+a[i];
return s;
}
	
	1
	n+1
	n
	1
		
		1
		1
		1
		1
			
			1
			n+1
			n
			1

	Total step count of algorithm =		2n+3

So, $t_{sum}(n)=2n+3$.

The step count tells us how the runtime of a program changes with the change in the instance characteristics.
→From the step count for the above algorithm sum, we see that, if n is doubled, the runtime also doubles(approximately).

EXAMPLE-2:-RSum algorithm:-
Assume the time complexity of Rsum is $t_{Rsum}(n)$.

Statement	No. of times of execution (frequency)		s/e	total no. of steps per statement	
	n<=0	n>0		n<=0	n>0

Algorithm Rsum(a,n)
{
if (n<=0) **then**
return 0.0;
else
return Rsum (a,n-1)+ a(n);
}
	
	1
	1
	
	0
		
		1
		0
		
		1
			
			1
			1
			
			$1+t_{Rsum}(n-1)$
				
				1
				1
				
				0
					
					1
					0
					
					$1 + t_{Rsum}(n-1)$

		Total Step Count =		2	$2+ t_{Rsum}(n-1)$

When analysing a recursive algorithm for its step count, we often obtain a recursive formula for the step count.
For the above algorithm,

$$t_{Rsum}(n) = \begin{cases} 2, & \text{if } n \le 0 \\ 2 + t_{Rsum}(n-1), & \text{if } n > 0 \end{cases}$$

→These recursive formulas are referred to as recurrence relations.
→One way of solving any such recurrence relation is to make repeated substitutions for each occurrence of the function or term t_{Rsum} on the RHS until all such occurrences disappear.
Solving the above recurrence relation:

$t_{Rsum}(n) = 2+t_{Rsum}(n-1)$
$t(n) = 2+t(n-1)$
$=2+(2+t(n-2))$
$=2+2+(2+t(n-3))$
$=2*3+t(n-3)$

After k substations,
$t(n)=2*k + t(n-k)$
If n=k, then
$t(n) = 2*n + t(0)$
$= 2n + 2$

$t_{Rsum}(n)=2n+2$

EXAMPLE-3:- Addition of two mxn matrices

Statement	No. of times of execution (frequency)	s/e	total no. of steps per statement

Algorithm Add(a,b,c,m,n)
{
for i:=1 to m **do**
  for j:=1 to n **do**
  c[i,j]:=a[i,j]+ b[i,j];
}
	
	m+1
	m(n+1)
	mn
		
		1
		1
		1
			
			m+1
			mn+ m
			mn

		Total step count =	2mn+2m+1

EXAMPLE-3a:- Addition of two nxn matrices

Statement	No. of times of execution (frequency)	s/e	total no. of steps per statement

Algorithm Add(a,b,c,n)
{
for i:=1 to n **do**
  for j:=1 to n **do**
  c[i,j]:=a[i,j]+ b[i,j];
}
	
	n+1
	n(n+1)
	n^2
		
		1
		1
		1
			
			n+1
			n^2+ n
			n^2

		Total step count =	$2n^2$+2n+1

EXAMPLE-3b:- Multiplication of two nxn matrices

Statement	No. of times of execution (frequency)	s/e	total no. of steps per statement

Algorithm Mul(a,b,c,n)
{
for i:=1 to n **do**
{
for j:=1 to n **do**
{
c[i,j]:=0;
for k:=1 to n **do**
{
c[i,j]:= c[i,j]+a[i,k]* b[k,j];
}
}
}
}
	
	n+1
	
	n(n+1)
	
	n^2
	n^2(n+1)
	
	n^3
		
		1
		
		1
		
		1
		1
		
		1
			
			n+1
			
			n^2+ n
			
			n^2
			n^3+ n^2
			
			n^3

		Total step count =	$2n^3$+$3n^2$+2n+1

EXAMPLE-4:- Algorithm which takes input n and computes the nth Fibonacci number and prints it.
->The Fibonacci sequence is: 0, 1, 1,2,3,5 . . .
->If we name the first term of the sequence as f_1 and second term as f_2, then f1=0 and f2=1, and in general,

$f_n=f_{n-1}+f_{n-2}$, where $n\geq 3$.

Statement	No. of times of execution (frequency)		s/e	total no. of steps per statement	
	n<=2	n>2		n<=2	n>2

Algorithm Fibonacci(n)
{
if(n≤2) **then**
write(n-1);
else
{
fn-2:=0; fn-1:=1;
for i:=3 to n **do**
{
fn:=fn-1+fn-2;
fn-2:=fn-1;
fn-1:=fn;
}
write(fn);
}
}
	
	1
	1
	
	0
	0
	
	0
	0
	0
	
	0
		
		1
		0
		
		1
		n-1
		
		n-2
		n-2
		n-2
		
		1
			
			1
			1
			
			2
			1
			
			1
			1
			1
			
			1
				
				1
				1
				
				0
				0
				
				0
				0
				0
				
				0
					
					1
					0
					
					2
					n-1
					
					n-2
					n-2
					n-2
					
					1

			Total step count =	2	4n-3

Order (or, Rate) of growth of running time:

→Our motivation to determine step count is to compare the running times of two alternative algorithms that perform the same task and also to predict how the running time of an algorithm grows as its input size changes.

So, would like to determine the order of growth of the running time of an algorithm (in terms of its input size), rather than determining its exact running time. For this purpose, we use asymptotic notations.

→The logic behind the above idea is as follows:
For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the higher-order term.

EXAMPLE:-
→Suppose the runtime of an algorithm is $6n^2+100n+300$.
The term $100n+300$ becomes less significant to the total value of the function as n grows larger. So, we can drop the term $100n+300$. And we are left with only $6n^2$. We can also drop the coefficient 6. And we can say that the running time of this algorithm grows in proportion to n^2.

Asymptotic notations:-
We will use asymptotic notations primarily to describe the running times of algorithms.

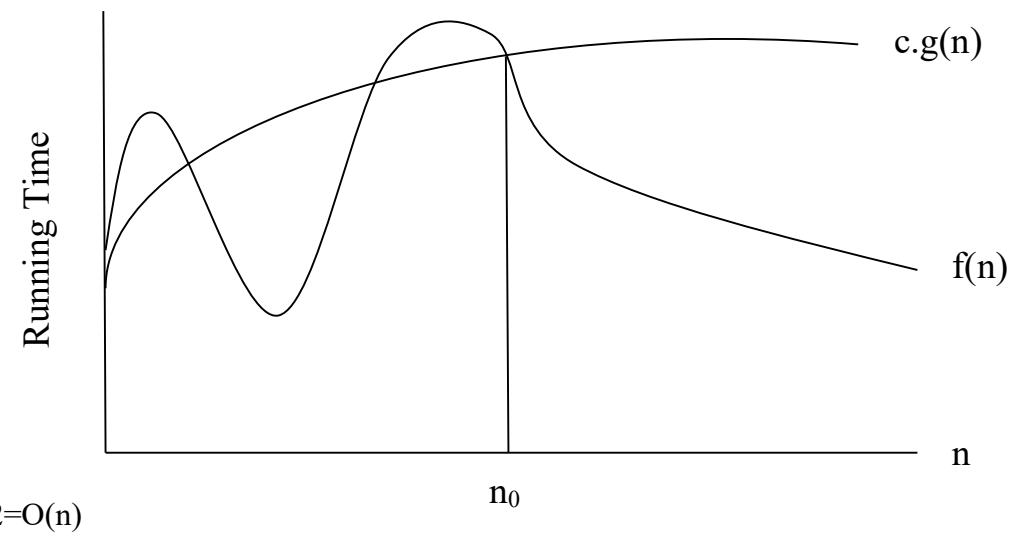
However, asymptotic notations actually apply to functions.

Asymptotic notation is used to describe the limiting behaviour of a function, when its argument tends towards a particular value (often infinity) usually in terms of simpler functions.

While analysing the runtime of an algorithm, we should not only determine how long the algorithm takes in terms of its inputs size but also should focus on how fast this runtime function grows with the input size which is facilitated by asymptotic notations.

(1) Big-Oh notation: - (O)

Let f(n) and g(n) be functions mapping non-negative integers to non-negative real numbers. We can say that f(n) is O(g(n)) iff there exists a real constant c>0 and an integer constant $n_0>0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.



Ex:-
1) $3n+2=O(n)$
Here,
$f(n)=3n+2$
$g(n)=n$.
$3n+2=O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$, where $c=4$ and $n_0=2$.
2) $3n+3=O(n)$
3) $100n+6=O(n)$, as $100n+6 \leq 101n$, for all $n \geq 6$.
4) $10n^2+4n+2=O(n^2)$, as $10n^2+4n+2 \leq 11n^2$, for all $n \geq 5$. (or) $10n^2+4n+2 \leq 16n^2$, for all $n \geq 1$.
5) $6*2^n+n^2=O(2^n)$, as $6*2^n+n^2 \leq 7*2^n$, for all $n \geq 4$.
6) $3n+3=O(n^2)$, as $3n+3 \leq 3n^2$, for all $n \geq 1$ **(or)** $3n+3 \leq n^2$, for all $n \geq 4$
7) $2^{100}=O(1)$, as $2^{100} \leq 2^{100}.1$, for all $n \geq 1$.
Note:-We write O(1) to mean a computing time of constant.

Theorem:- If $f(n)=a_m n^m+.....+a_1 n+a_0$, then $f(n) = O(n^m)$.

Proof:-
$$f(n) = \sum_{i=0}^{m} a_i n^i$$
$$\leq \sum_{i=0}^{m} |a_i| n^i = n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$
$$\leq n^m \sum_{i=0}^{m} |a_i|, \text{ for all } n \geq 1.$$
$$\leq c.g(n), \text{ where } c=\sum_{i=0}^{m} |a_i| \text{ and } g(n)= n^m.$$
So, $f(n) = O(n^m)$.

Ex:- $10n^2+4n+2=O(n^2)$ as $10n^2+4n+2 \leq 16n^2$ for all $n \geq 1$.
→There are several functions g(n) for which f(n)=O(g(n)) is true. The statement f(n)=O(g(n)) states that g(n) is only an upper bound on the value of f(n) for all $n \geq n_0$. For the statement f(n)=O(g(n)) to be meaningful, g(n) should be as small function as possible (i.e., least upper bound) for which f(n)=O(g(n)) is true.

So, while we often say that, $3n+3=O(n)$ and $10n^2+4n+2=O(n^2)$, we almost never say that, $3n+3=O(n^2)$ or $10n^2+4n+2=O(n^3)$, even though both of these statements are true.

Frequently used Efficiency classes:
→O(1) is called Constant time.
→O(n) is called Linear time.
→$O(n^2)$ is called Quadratic time.
→$O(n^3)$ is called Cubic time.
→$O(n^k)$, k>=1; is called Polynomial time.
→$O(2^n)$ and $O(a^n)$ are called Exponential time.
→O(log n) is called Logarithmic time.

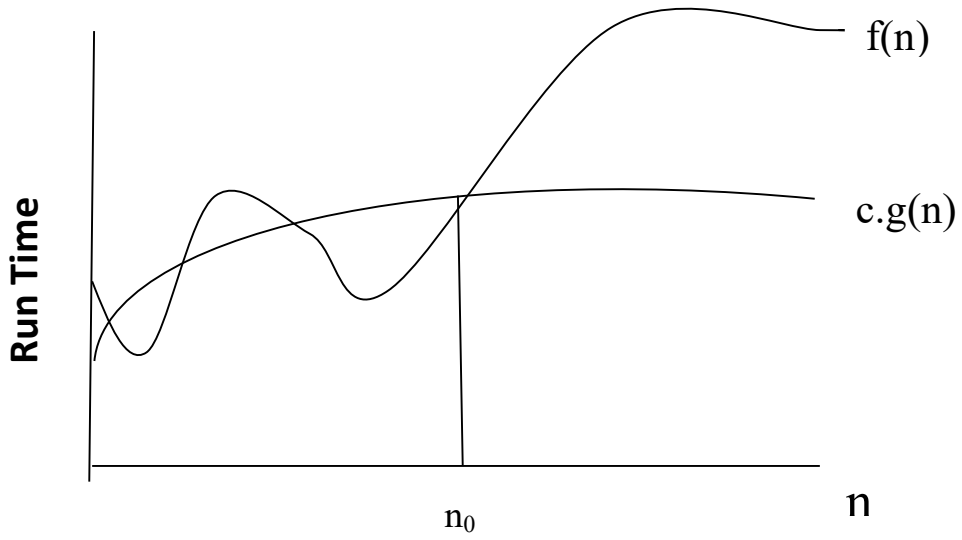
→For sufficiently large values of n, the following relationship holds among efficiency classes:
$Constant<logn<n<nlogn<n^2<n^3<2^n<n!$

Ex:- $n^2+nlogn+n+4=O(n^2)$.
$3n+2 \neq O(1)$, as $3n+2$ is not less than or equal to any constant c for all $n \geq n_0$.
$10n^2+4n+2 \neq O(n)$.

Problem1: (GATE-2017 Set1)
Consider the following functions from positives integers to real numbers:
10, √n, n, log_2n, 100/n.
The CORRECT arrangement of the above functions in increasing order of asymptotic complexity is:
(A) log_2n, 100/n, 10, √n, n
(B) 100/n, 10, log_2n, √n, n
(C) 10, 100/n ,√n, log_2n, n
(D) 100/n, log_2n, 10 ,√n, n
NOTE:-
1)$n!=O(n^n)$
2)$\log n^3=O(logn)$.

2)Big-omega notation(Ω):-

Let f(n) and g(n) be functions mapping non-negative integers to non-negative real numbers. We say that f(n) is Ω(g(n)) iff there exists a real constant c>0 and an integer constant $n_0>0$ such that $f(n) \geq c.g(n)$ for all $n \geq n_0$.

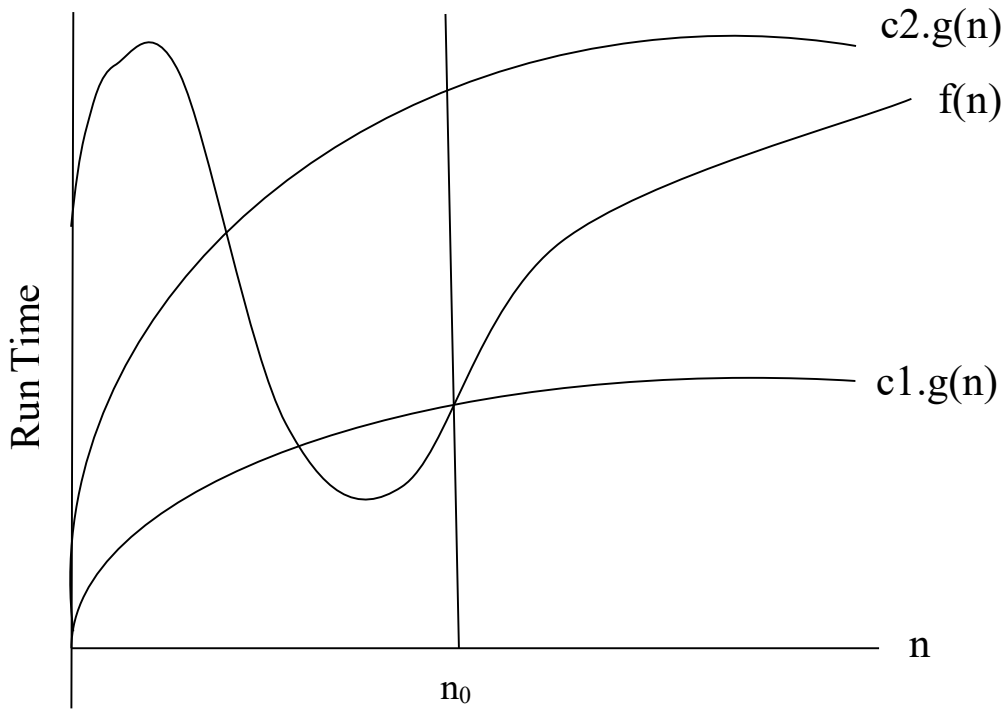


Ex:-
1)$3n+2=\Omega(n)$as, $3n+2 \geq 3n$ for all $n \geq 1$ where c=3 and $n_0=1$.
2)$3n+3=\Omega(n)$ as,$3n+3 \geq 3n$ for all $n \geq 1$.
3)$100n+6=\Omega(n)$ as, $100n+6 \geq 100n$ for all $n \geq 1$.
4)$10n^2+4n+2=\Omega(n^2)$ as, $10n^2+4n+2 \geq n^2$ for all $n \geq 1$, where c=1 and $n_0=1$.
5)$6*2^n+n^2=\Omega(2^n)$ as, $6*2^n+n^2 \geq 2^n$ for all n>=1.
6)$3n+3=\Omega(1)$ since $3n+3 \geq 1$ for all n>=1
but $3n+3 \neq O(1)$.
7)$10n^2+4n+2= \Omega(n^2)=\Omega(n)= \Omega(1)$.

There are several functions g(n) for which $f(n)=\Omega(g(n))$ is true. The function g(n) is only a lower bound on f(n). For the statement, $f(n)=\Omega(g(n))$ to be meaningful, g(n) should be as large function as possible (i.e., largest lower bound) for which the statement, $f(n)=\Omega(g(n))$ is true.
→So, while we say that, $3n+3=\Omega(n)$ and $10n^2+4n+2=\Omega(n^2)$, we almost never say that $3n+3=\Omega(1)$ or $10n^2+4n+2=\Omega(n)$ even though both of these statements are correct.

Big-Theta notation(Θ):-

Let f(n) and g(n) be functions mapping non-negative integers to non-negative real numbers. We say that f(n) is Θ(g(n)) iff there exist two real constants c1>0 and c2>0 and an integer constant $n_0>0$ such that $c1.g(n) \leq f(n) \leq c2.g(n)$ for all $n \geq n_0$. That means, f(n)= Θ(g(n)) iff f(n)=O(g(n)) and $f(n)=\Omega(g(n))$.
We can say that f(n)= Θ(g(n)) iff g(n) is both an upper bound and lower bound on f(n).



Ex:-
1)$3n+2=\Theta(n)$ as, $3n+2 \geq 3n$ and $3n+2 \leq 4n$ for all $n \geq 2$, where g(n)=n, c1=3, c2=4 and $n_0=2$.

2)3n+3=Θ(n) since 3n+3= O(n)= Ω(n)
3)$10n^2$+4n+2=Θ(n^2).
4)6*2^n+n^2=Θ(2^n).
5)10*logn+4=Θ(logn).
6)3n+2≠Θ(1) since 3n+2= Ω(1) but 3n+2≠ O(1)
7)3n+3≠Θ(n^2).
8)$10n^2$+4n+2≠Θ(n) and
$10n^2$+4n+2≠Θ(1).
9)6*2^n+n^2≠Θ(n^2) and 6*2^n+n^2≠Θ(1).

Little-oh naotation(o):- (f(n)<g(n))
The function f(n)=o(g(n)) iff $\lim_{n\to\infty} f(n)/g(n)=0$.
Ex:-
1)3n+2=o(n^2) since $\lim_{n\to\infty}(3n+2)/n^2=0$. [since (3/n)+(2/$n^2$)=0 when n=∞].
2)3n+2=o(nlogn) since $\lim_{n\to\infty}((3n+2)/nlogn)=\lim_{n\to\infty}\{\left(\frac{3}{logn}\right)+\left(\frac{2}{nlogn}\right)\}=0$.
3)6*2^n+n^2=o(3^n) since $\lim_{n\to\infty}(6*2^n+n^2)/(3^n)=\lim_{n\to\infty}\{\frac{6*2^n}{3^n}+n^2/3^n\}=0$.
$=\lim_{n\to\infty}\{6*(\frac{2}{3})^n+n^2/3^n\}=\lim_{n\to\infty}\{n^2/3^n\}$
Using L'hopital's rule: (i.e., taking derivatives on both numerator and denominator)
$\lim_{n\to\infty}\{n^2/3^n\}=\lim_{n\to\infty}\{2n/(ln(3)3^n)\}=\lim_{n\to\infty}\{(2/ln(3))*1/(ln(3)3^n)\}=0$

4)3n+2≠o(n) since $\lim_{n\to\infty}(3n+2)/n=\lim_{n\to\infty}3+(\frac{2}{n})=3\neq 0$.

Little omega notation(ω):- (f(n)>g(n))
The function f(n)=ω(g(n)) iff $\lim_{n\to\infty} g(n)/f(n)=0$.
Ex:-
1) 3n+2=ω(1) since $\lim_{n\to\infty} 1/(3n+2)=0$.
2) $10n^2$+4n+2 = ω(n) since $\lim_{n\to\infty} n/(10n^2+4n+2)=0$.
$=\omega(1)$
$\neq\omega(n^2)$

Problem: - (GATE-2015 Set3 Question)

Consider the equality $\sum_{i=0}^{n} i^3 = X$ and the following choices for X

I. $\Theta(n^4)$
II. $\Theta(n^5)$
III. $O(n^5)$
IV. $\Omega(n^3)$

The equality above remains correct if X is replaced by

(A) Only I
(B) Only II
(C) I or III or IV but not II
(D) II or III or IV but not I

Hint:

$$\sum_{1\le i\le n} i^k = \frac{n^{k+1}}{k+1} + \frac{n^k}{2} + \text{lower order terms}$$

Problem: - (GATE-2015 Set3 Question)

Let f(n) = n and g(n) = $n^{(1+\sin n)}$, where n is a positive integer. Which of the following statements is/are correct?
I. f(n) = O(g(n))
II. f(n) = Ω(g(n))
(A) Only I
(B) Only II
(C) Both I and II
(D) Neither I nor II

Answer: (D)

Explanation: The value of sine function varies from -1 to 1.
For sin = -1 or any other negative value, I becomes false.
For sin = 1 or any other positive value, II becomes false.

Asymptotic Notations and Time Complexities of Algorithms:
Problem-1:-

Give a big-O characterization in terms of n, of the running time of the following code.

```
Sum:=0                  ------------------->1
for i:=1 to n do        --------->n+1
sum= sum+i;             ---------->n
```

Running time=2n+2= O (n).

Problem-2:-

Give a big oh characterization in terms of n of the running time of the loop 1 method shown in the following algorithm.

```
Algorithm loop1(n)
{
p:=1                        ------------------------->1
for i:=1 to 2n do           ---------->2n+1
  p:=p*i                    ------------------------->2n
}
```

Problem-3:- (C code fragment)

```
Algorithm loop2(n)
{
for(i=n; i>=1;)
{
  i=i/2;
  print(i);
}
```

Time complexity = Number of times the loop executed.
Initial value of i is n and final value of i is 1.
If the loop is executed x times, then $n/2^x = 1 \Rightarrow n = 2^x \Rightarrow x = \log_2 n \Rightarrow x = O(\log n)$

Problem-4:- (C code fragment)

```
Algorithm loop3(n)
{
for(j=1; j<=n;)
{
  j=j*2;
  print(j);
}
```

Time complexity = Number of times the loop executed.
Initial value of j is 1 and final value of j is n.
If the loop is executed x times, then $2^x = n \Rightarrow x = \log_2 n \Rightarrow x = O(\log n)$

Problem-5:- (C code fragment)

```
i=1;
S=0;
while(S<=n)
{
        S=S+i;
        i++;
}
```

Time complexity = Number of times the loop executed.
Initial value of S is 0 and final value of S is n.
If the loop is executed k times, then final value of S= 0+1+2+3+...+k = n
$k(k+1)/2 = n \Rightarrow k = O(\sqrt{n})$.

Practice Problem:

What is the complexity of the following code?

```
1. sum=0;
2.    for(i=1;i<=n;i*=2)
3.        for(j=1;j<=n;j++)
4.            sum++;
```

A. $O(n^2)$
B. O(n logn)
C. O(n)
D. O(n logn logn)

Practice Problem:

Consider the following C function.

```
int fun(int n)
{
   int i, j;
   for (i = 1; i <= n ; i++)
   {
      for (j = 1;  j < n; j += i)
      {
         printf("%d %d", i, j);
      }
   }
}
```

Time complexity of fun in terms of θ notation is:
(A) $\theta(n \sqrt{n})$
(B) $\theta(n^2)$
(C) $\theta(n \log n)$
(D) $\theta(n^2 \log n)$

Practice Problem (GATE 2015 Set-1)
Consider the following C function.

```
int fun1 (int n)
{
int i, j, k, p, q = 0;
for (i = 1; i<n; ++i)
{
        p = 0;
        for (j = n; j > 1; j = j/2)
                ++p;
        for (k = 1; k < p; k = k*2)
                ++q;
}
return q;
}
```

Which one of the following most closely approximates the return value of the function fun1?
(A) n^3
(B) $n (\log n)^2$
(C) nlogn
(D) nlog(logn)

Practice Problem (GATE 2013)
Consider the following function:

```
int unknown(int n) {
        int i, j, k = 0;
        for (i = n/2; i <= n; i++)
                for (j = 2; j <= n; j = j * 2)
                        k = k + n/2;
        return k;
}
```

The return value of the function is
A. $\Theta(n^2)$
B. $\Theta(n^2 \log n)$
C. $\Theta(n^3)$
D. $\Theta(n^3 \log n)$

Problem-6: Present an algorithm that searches an unsorted array a[1: n] for the element x. If x is present then return a position in the array; else return 0. And analyse its time complexity.
Sol:

```
Algorithm  Search(a,n,x)
{
   for i:=1 to n do
      if(a[i]=x) then
            return i;
   return 0;
}
```

The above algorithm may terminate in one iteration (i.e., 3 steps) if x is present in the first position, or it may take two iterations (i.e., 5 steps) if x is present in the second position, and so on.

In other words, knowing 'n' alone is not enough to estimate the runtime of the algorithm.

How to overcome this difficulty in determining the step count uniquely?

Explanation: -
When the chosen parameters are not adequate to determine the step count (or, time complexity) uniquely, we define 3 kinds of step counts (or, time complexities) :
i.e., **Best case**
Worst case
Average case

Best case step count (or, Best case Time complexity):-
It is the minimum number of steps taken by the algorithm for *any* input of size n.
(OR)
It is smallest running time of the algorithm for *any* input of size n.

Worst case step count (or, Worst case Time complexity):-
It is the maximum number of steps taken by the algorithm for *any* input of size n.
(OR)
It is longest running time of the algorithm for *any* input of size n.

Average case step count (or, Average case Time complexity):-
It is the average number of steps taken by the algorithm on all instances of input with size n.
(OR)
It is running time of the algorithm for a random instance of input of size n.

→For the above algorithm, the best-case time complexity happens when the element x is present in the first position and the worst-case time complexity happens when the element x is present in the last position or if it is not present.

Statement	**Best case** (Assuming that the element x is present in the first position)	**Worst case** (Assuming that the element x is present in the last position)
	total no. of steps per statement	**total no. of steps per statement**

Algorithm Search(a,n,x)
{
for i:=1 **to** n **do**
if(a[i]=x) **then**
return i;
return 0;
}
	
	1
	1
	1
	0
		
		n
		n
		1
		0

Total step count =	3	2n+1

The best-case step count = 3 =Constant
= O(1) = Ω(1) = Θ(1)

The worst-case step count = 2n+1
= O(n) = Ω(n)= Θ(n)

The average case step count = (3+5+7+...+2n+1)/n = (3+5+7+...+2n-1+2n+1)/n
= ((1+3+5+7+...+2n-1)+2n)/n
= (n^2+2n)/n
= n+2
= O(n) = Ω(n) = Θ(n)

Average Time complexity ≤ Worst-case Time complexity.

Note:
1. The worst-case running time of an algorithm gives us an upper bound on the running time of the algorithm for any input. We use 'O' notation to denote the upper bound on the running time of an algorithm.

2. The best-case running time of an algorithm gives us a lower bound on the running time of the algorithm for any input. We use 'Ω' notation to denote the lower bound on the running time of an algorithm.

3. We use 'Θ' notation to denote the running time of an algorithm if both the upper and lower bounds on the running time of the algorithm are same.

4. We generally concentrate on upper bound because knowing lower bound of an algorithm is of no practical importance.

So, we can say that the time complexity of the linear search algorithm is O(n) because it gives the upper bound on the run time (i.e., it indicates the maximum run time).

It is also true that the time complexity of the linear search algorithm is Ω(1) because it gives the lower bound on the run time (i.e., it indicates the minimum run time).

But generally, we don't express the time complexity of an algorithm, alone in terms of the lower bound on its run time because it doesn't give any information about the upper bound on the run time (i.e., maximum running time) of the algorithm which is of prime importance for us.

Problem 7: (GATE-2007)

Consider the following C code segment:

```
int IsPrime(n)
{
  int i, n;
  for(i=2; i<=sqrt(n); i++)
     if(n%i == 0)
       {printf("Not Prime\n"); return 0;}
  return 1;
}
```

Let T(n) denotes the number of times the for loop is executed by the program on input n. Which of the following is TRUE?
(A) T(n) = O(√n) and T(n) = Ω(√n)
(B) T(n) = O(√n) and T(n) = Ω(1)
(C) T(n) = O(n) and T(n) = Ω(√n)
(D) None of the above

Problem 8: (GATE-2013)

Which one of the following is the tightest upper bound that represents the number of swaps required to sort n numbers using selection sort?
(A) O(log n)
(B) O(n)
(C) O(nLogn)
(D) O(n^2)

Problem: Analyze the Time complexity of Insertion Sort

```
Algorithm InsertionSort (a, n)
{
        // sort the array a[1:n] into non decreasing order.
         for  i := 2 to n do
         {
                key:=a[i];
                // Insert a[i] into the sorted part of the array, i.e., a [1: i-1]
                    j:=i-1;
                while(j≥1 and a[j]>key) do   //Searching for the position of key and inserting it in its place by shifting
                                                    //the larger elements right side
                {
                        a[j+1]:=a[j];  // move a[j] to its next position  in the right side
                        j:=j-1;
                }
                a[j+1]:=key;
         }
}
```

In Worst-case (when the elements are in descending order), the time complexity is $O(n^2)$.
In Best-case (when the elements are in ascending order), the time complexity is O(n).
In Average-case, the time complexity is $O(n^2)$.

Disjoint Sets

Suppose we have some finite universe of n elements, U, out of which sets will be constructed. These sets may be empty or contain any subset of the elements of U. We shall assume that the elements of the sets are the numbers 1, 2, 3, …, n.

We assume that the sets being represented are pair wise disjoint, i.e., if S_i and S_j ($i \neq j$) are two sets, then there is no element that is in both $\mathbf{S_i}$ and $\mathbf{S_j}$.

For example, when n = 10, the elements can be partitioned into three disjoint sets, $S_1 = \{1,7, 8, 9\}$, $S_2 = \{2,5,10\}$, and $S_3 = \{3,4,6\}$.

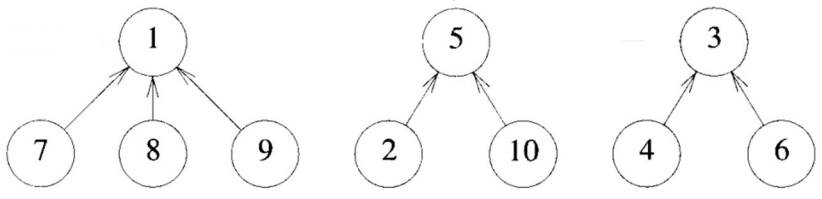
→The following two operations are performed on the disjoint sets:
1) Union:
If $\mathbf{S_i}$ and $\mathbf{S_j}$ are two disjoint sets, then their union $\mathbf{S_i}$ U $\mathbf{S_j}$ = {all elements **x** such that **x** is in $\mathbf{S_i}$ or $\mathbf{S_j}$}. Thus, in our example, $\mathbf{S_1}$ U $\mathbf{S_2}$ = {1, 7, 8, 9, 2, 5, 10}.

Since we have assumed that all sets are disjoint, we can assume that following the union of $\mathbf{S_i}$ and $\mathbf{S_j}$, the sets $\mathbf{S_i}$ and $\mathbf{S_j}$ do not exist independently; that is, they are replaced by $\mathbf{S_i}$ U $\mathbf{S_j}$ in the collection of sets.

2) Find(i): Given the element i, find the set containing i.
Thus, in our example, 4 is in set S_3, and 9 is in set S_1.
So, Find(4) = S_3
Find(9) = S_1

To carry out these two operations efficiently, we represent each set by a tree.

One possible representation for the sets $S_1 = \{1,7, 8,9\}$, $S_2 = \{2,5,10\}$, and $S_3 = \{3,4,6\}$ using trees is given below:



Note: For each set, we have linked the nodes from the children to the parent.

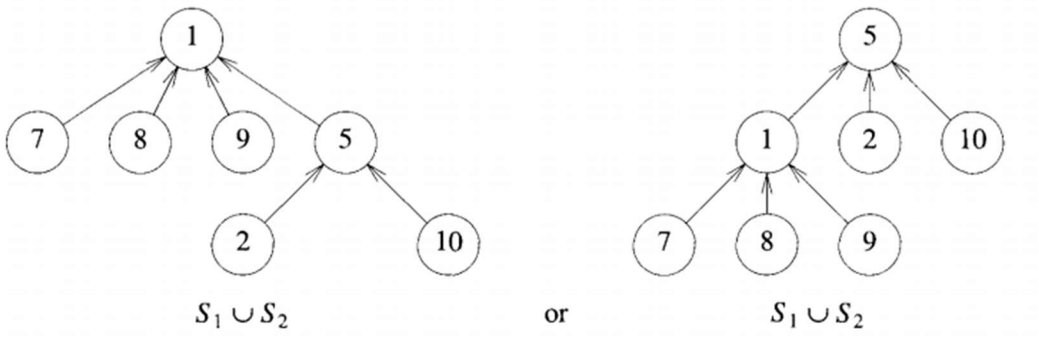
In presenting the UNION and FIND algorithms, we ignore the set names and identify sets just by the roots of the trees representing sets. This simplifies the discussion.

The operation of Find(i) now becomes:
Determine the root of the tree containing element 'i'.

Ex: Find(1)=1, Find(7)=1, Find(5)=5, Find(2)=5, Find(3)=3, find(6)=3, and so on.

To obtain the union of two sets, all that has to be done is to link one of the roots to the other root. The function Union(i, j) requires two trees with roots i and j to be joined.

The possible representations of $S_1 \cup S_2$:



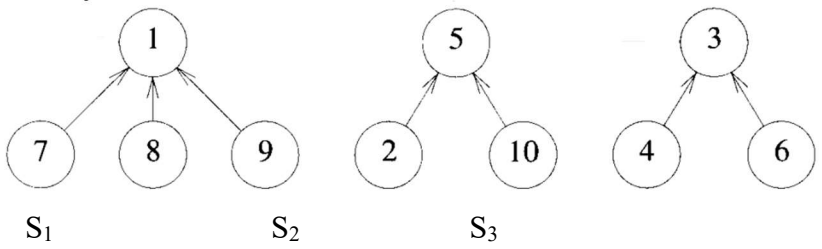
Representing the tree nodes of all disjoint sets using a single array:

Since the universal set elements are numbered 1 through n, we represent the tree nodes of all sets using an array P[1:n], where P stands for parent.

The i^{th} index of this array represents the tree node for element *i*. The array element at index *i* gives the parent of the corresponding tree node.

Note: We assume that the parent of root node of disjoint set tree is -1.

Ex: Suppose the tree representations of disjoint sets S_1, S_2 and S_3 are as follow:



Array representation of trees corresponding to sets S_1, S_2 and S_3:

i	1	2	3	4	5	6	7	8	9	10
P[i]	-1	5	-1	3	-1	3	1	1	1	5

→We can now implement **Find(i)** by following the indices starting at i until we reach a node with parent value -1.

Simple algorithm for **FIND(i)**

```
Algorithm SimpleFind(i)
{
        while (P[i] ≥ 0) do
                i := P[i];
        return i;
}
```

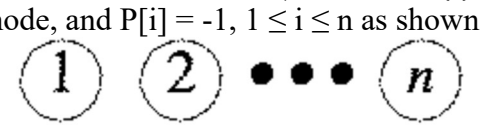
→The operation **Union(i, j)** is equally simple. Adopting the convention that the first tree becomes a subtree of the second tree (i.e., root of the first tree is linked to the root of the second tree), the statement P[i] := j accomplishes the Union.

Simple algorithm for **Union (i, j)**

```
Algorithm SimpleUnion(i, j)
{
        P[i] := j;
}
```

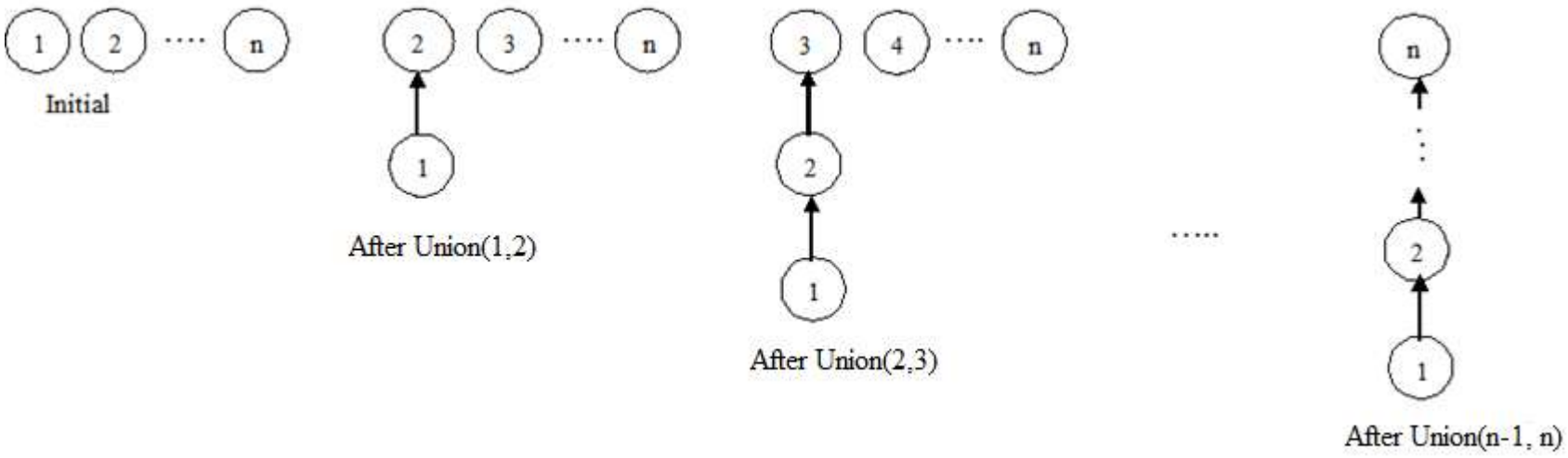
→Although these two algorithms are very easy to state, their performance characteristics are not very good.

For example, if we start off with 'n' elements each in a set of its own (that is, $S_i = \{i\}$, $1 \le i \le n$), then the initial configuration consists of a forest with 'n' trees each consisting of one node, and P[i] = -1, $1 \le i \le n$ as shown below:



i	1	2	…	n-1	N
P[i]	-1	-1	…	-1	-1

➔ Now imagine that we process the following sequence of UNION operations in the worst case:
Union(1,2), Union(2,3), …, Union(n-1, n).



This sequence of union operations results in the **degenerate tree** as shown above.

The time taken for a union operation is constant and so, the *n-1* Union operations can be processed in **O(n)** time.

→Now suppose we process the following sequence of FIND operations:

Find(1), Find(2), …, Find(n).

Each FIND requires following a chain of the parent links from the element to be found up to the root.

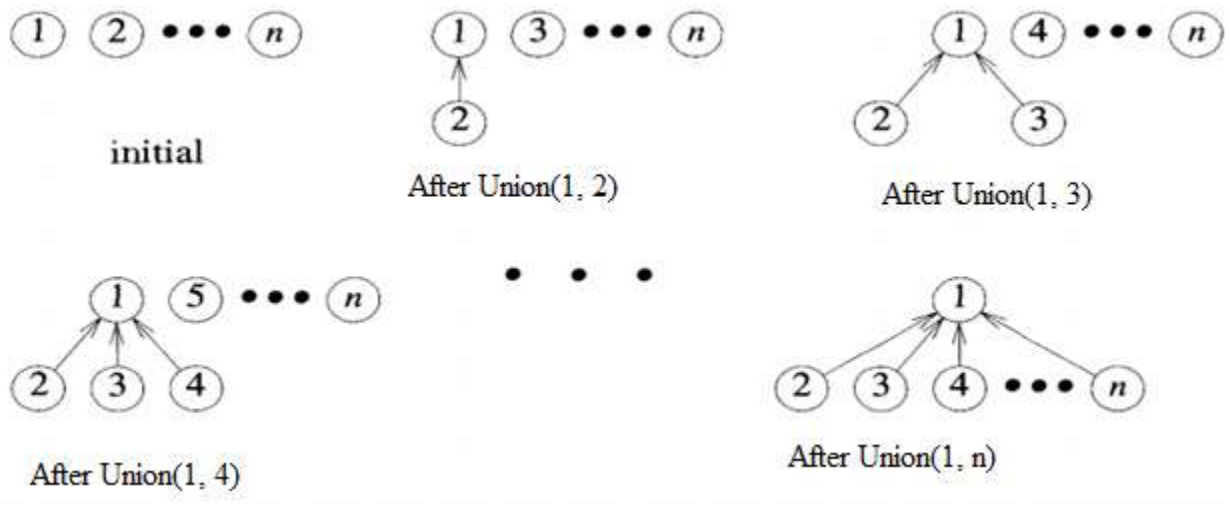
Since the time required to process a FIND for an element at level 'i' of the tree is O(i), the total time needed to process the 'n' FIND operations is $\sum_{i=1}^{n} i = \mathbf{O(n^2)}$.

We can improve the performance of our UNION and FIND algorithms by avoiding the creation of degenerate trees. To accomplish this, we make use of a weighting rule for Union(i, j).

Weighting rule for Union(i, j):

If the number of nodes in the tree with root *i* is less than the number of nodes in the tree with root *j*, then make *'j'* as the parent of *'i'*; otherwise make *'i'* as the parent of *'j'*.

→When we use the weighting rule to perform the sequence of UNION operations Union(1,2), Union(1,3), …, Union(1,n), we obtain the trees as shown below:



To implement the weighting rule, we need to know how many nodes are there in every tree. To do this easily, we maintain a count field in the root of every tree. If *'i'* is a root node, then *count[i]* = number of nodes in the tree.

Since all nodes other than the roots of the trees have a positive number in their corresponding positions in the P[ ] array, we can maintain the negative of count of a tree in the corresponding position of its root in P[ ] array to distinguish root from other nodes.

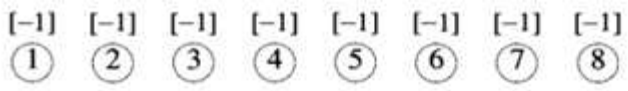
Union algorithm with weighting rule :

```
Algorithm WeightedUnion(i, j)
{
        // Unite sets with the roots i and j (i ≠ j) using weighting rule.
        // P[i] = -count[ i ] and P[ j ] = -count[ j ]
        temp := P[ i ] + P[ j ];
        if(P[ i ] > P[ j ]) then    // if tree 'i' has lesser number of nodes than tree 'j'
        {
                P[i ] := j;    // make i as subtree of j
                P[ j ] := temp;       //update count of  tree j
        }
        else   // if tree 'i' has more or same number of nodes than tree 'j'
        {
                P[ j ] := i;   // make j as subtree of  i
                P[ i ] := temp;      //update count of  tree i

        }
}
```

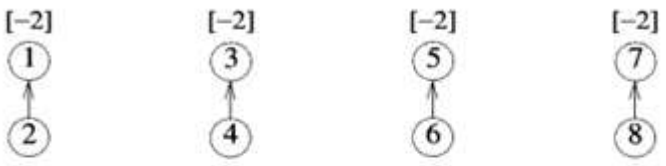
The time taken for WeightedUnion(i, j) is also constant, that is, **O(1)**.

Note:
Assume that we start off with a forest of trees, each having one node. Let *T* be a tree with *'n'* nodes created as a result of a sequence of UNION operations each performed using WeightedUnion. The height of *T* will not be more than$\lfloor \log_2 n \rfloor$. So, the worst-case time complexity of FIND is **O(logn)**.

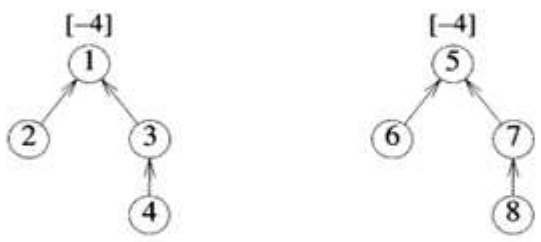
Example:
Consider the behavior of WeightedUnion on the following sequence of UNION operations starting from the initial configuration, P[i] = - count[i] = -1, 1≤i≤ 8 :
Union(1,2), Union(3,4), Union(5,6), Union(7,8), Union(1,3), Union(5,7), Union(1,5):



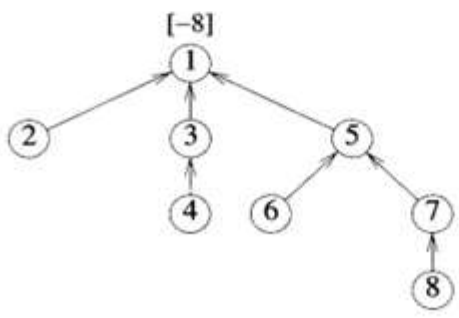
(a) Initial height-0 trees



(b) Height-1 trees following *Union*(1,2), (3,4), (5,6), and (7,8)



(c) Height-2 trees following *Union*(1,3) and (5,7)



(d) Height-3 tree following *Union*(1,5)

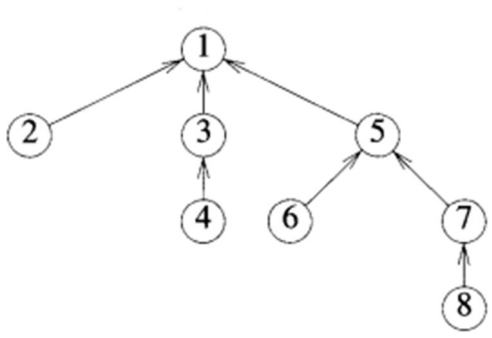
To further reduce the time taken over a sequence of FIND operations, we make the modifications in the FIND algorithm using the Collapsing Rule.

Collapsing Rule :
If 'j' is a node on the path from 'i' to its root 'r' and P[i] ≠ r then set P[j] = r.

Find algorithm with Collapsing Rule :

```
Algorithm CollapsingFind(i)
{
      // Find the root of the tree containing element i. Use the collapsing rule to collapse all nodes from i to root.
        r :=i;
      while(P[r] > 0) do        // find the root of the tree containing i.
            r := P[ r ];
      // At this point, r is the root of the tree containing i. Now collapsing has to done.
        while (i ≠ r) do
        {
            S := P[ i ] ;
            P[ i ] := r;   // link i to r directly
            i := S;
      }
      return r;
}
```

Example: Consider the following tree:

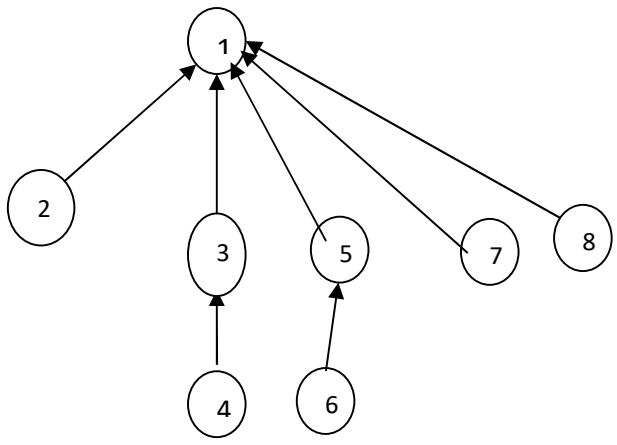


i	1	2	3	4	5	6	7	8
P [i]	-8	1	1	3	1	5	5	7

Now process the following eight FIND operations:
Find(8), Find(8), Find(8), Find(8), Find(8), Find(8), Find(8), Find(8).

If SimpleFind( ) is used, each Find(8) requires going up 3 parent link fields for a total of 24 moves to process all the eight FIND operations.

When CollapsingFind( ) is used, the first Find(8) requires going up 3 parent links and the resetting of 3 parent links. The tree after performing the first Find(8) operation will be as follows:

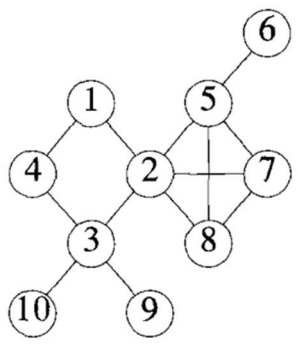


Each of the remaining seven Find(8) operations require going up only one parent link field. The total cost is now only 3+3+7=13 moves.

Articulation Points

A vertex V in a connected graph G is said to be an articulation point if and only if the deletion of vertex V together with all edges incident to V disconnects the graph into two or more non-empty components.

Example: Consider the following connected graph G:

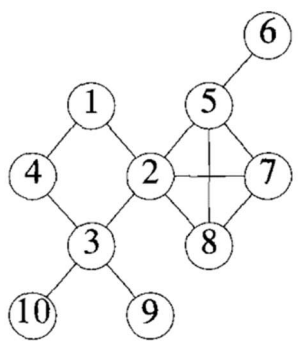


The articulation points in the graph G are: 2, 3 and 5.

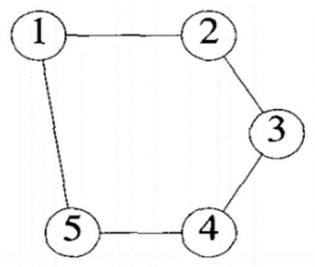
Biconnected Graph: A graph G is biconnected if and only if it contains no articulation points.

Examples:

1. The following graph is not a biconnected graph since it has articulation points.



2. The following graph is a Biconnected Graph since it doesn't have articulation points.



→The presence of articulation points in a connected graph can be undesirable feature in many cases.
For example, if G represents a communication network with the vertices representing communication stations and the edges representing communication lines, then the failure of a communication station i that is an articulation point would result in the loss of communication to other points also and makes the entire communication system down.

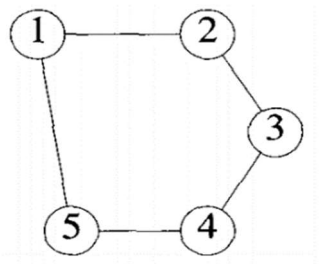
On the other hand, if G has no articulation point, then if any station i fails, we can still communicate between any two stations excluding station i.

Once it has been determined that a connected graph *G* is not biconnected, it may be desirable to determine a set of edges whose inclusion will make the graph biconnected. Determining such a set of edges is facilitated if we know the maximal subgraphs of *G* that are biconnected, (i.e., biconnected components of G).

Biconnected Components:

A biconnected component of a graph G is a maximal subgraph of G that is biconnected. That means, it is not contained in any larger subgraph of G that is biconnected.

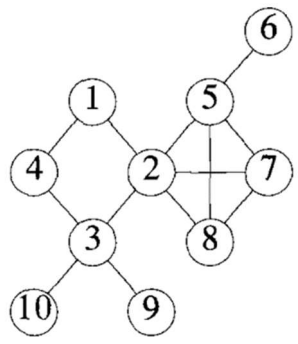
Ex: Consider the following biconnected graph:



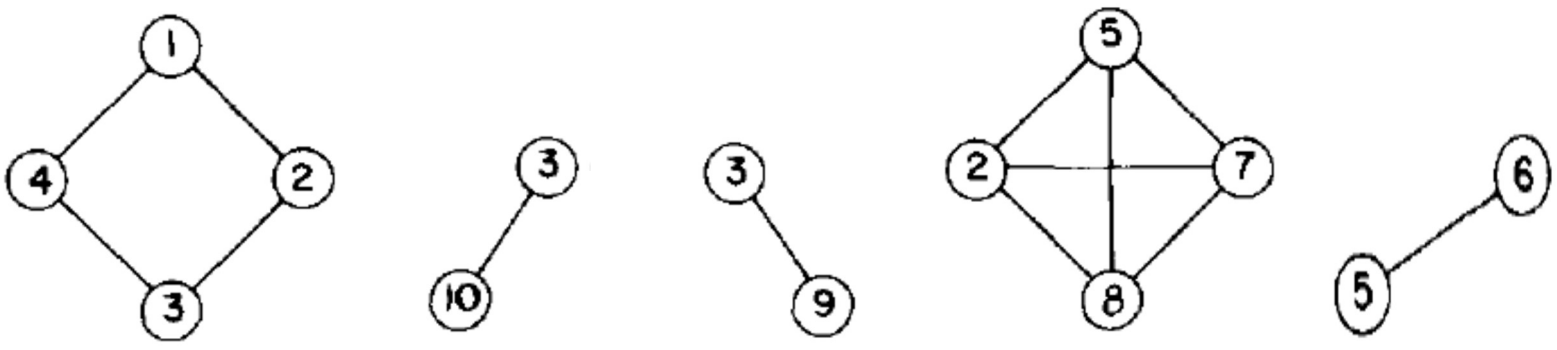
This graph has only one biconnected component (i.e., the entire graph itself).

So, a biconnected graph will have only one biconnected component, whereas a graph which is not biconnected consists of several biconnected components.

Ex: Consider the following graph which is not biconnected:



Biconnected components of the above graph are:



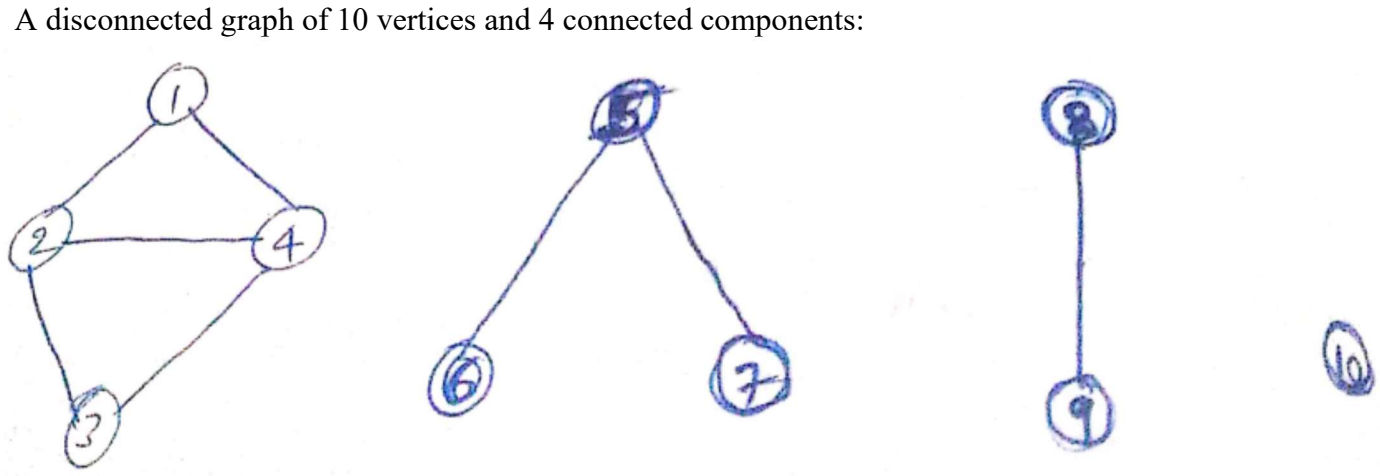
Note: Two biconnected components can have at most one vertex in common and this vertex is an articulation point.

Connected Components:

A connected component of a graph G is a maximal subgraph of G that is connected. That means, it is not contained in any larger subgraph of G that is connected.

A connected graph consists of just one connected component (i.e., the entire graph), whereas a disconnected graph consists of several connected components.

Ex: A disconnected graph of 10 vertices and 4 connected components:



UNIT-II

DIVIDE-AND-CONQUER

GENERAL METHOD: -

Divide-and-conquer algorithms work according to the following general plan:

1) **Divide:**
   A problem is divided into a number of subproblems (which are smaller instances of the given problem) of the same type, ideally of about equal size.
2) **Conquer:**
   The subproblems are solved (typically recursively). However, if the subproblems are small enough, they are solved in a straightforward manner.
3) **Combine:**
   If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

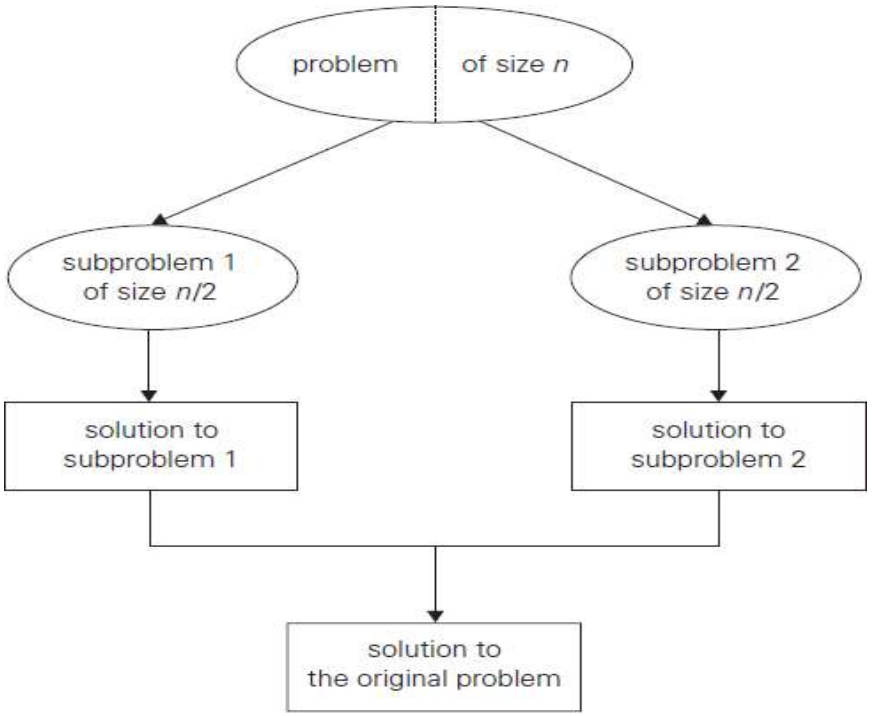


Figure: Divide-and-conquer technique (typical case)

Control abstraction for divide and conquer: -

```
1. Algorithm DandC(P)
2. {
3.      if Small(P) then
4.          return S(P);
5.      else
6.      {
7.          Divide P into smaller instances P1, P2, ..., Pk, where k ≥1;
8.          Apply DandC to each of these k sub-problems;
9.          return Combine(DandC(P1), DandC(P2), ..., DandC(Pk));
10.     }
11.}
```

→ Small(P) is a Boolean-valued function that determines whether the input size is small enough to compute without splitting and if so, the function S(P) is invoked. Otherwise, the problem P is divided into smaller sub-problems.
→ Combine( ) is a function that determines the solution to P using the solutions to the k sub-problems P1, P2, …, Pk.

Computing the time complexity of DandC:-

→ If the size of the problem P is *n* and the sizes of the *k* sub-problems are $n_1, n_2, \ldots, n_k$ respectively, then the computing time of DandC is described by the recurrence relation:

$$T(n) = \begin{cases} g(n), & if\ n\ is\ small \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n), & otherwise \end{cases}$$

where,

- *T(n)* is the time for DandC on any input of size *n*
- *g(n)* is the time to compute the answer directly for small inputs
- *f(n)* is the time taken for dividing P into sub-problems and combining the solutions of sub-problems.

→ The time complexity of many divide-and-conquer algorithms is given by recurrences of the form:

$$T(n) = \begin{cases} c, & if\ n = 1 \\ a\,T\left(n/b\right) + f(n), & if\ n > 1 \end{cases}$$

where a, b and c are constants.

We assume that *n* is a power of *b* (i.e., $n = b^k$, for some $k \geq 1$).

Eg :-1. Solve the following recurrence relation using substitution method:

$$T(n) = \begin{cases} 1, & if\ n = 1 \\ 2\,T(n/2) + n, & otherwise \end{cases}$$

$T(n) = 2T(n/2) + n$

$= 2[2T(n/4) + n/2] + n = 4T(n/4) + 2n$

$= 4[2T(n/8) + n/4] + 2n = 8T(n/8) + 3n = 2^3T(n/2^3) + 3n$

After applying this substitution k times,

$T(n) = 2^kT(n/2^k) + kn$

To terminate this substitution process, we switch to the closed form $T(1)=1$, which happens when $2^k = n$ which implies $k=\log_2 n$

So, $T(n) = nT(1) + n\log_2 n$

$T(n) = n + n\log n$

$= O(n\log n)$

Note:

(1) $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$

(2) $1 + r + r^2 + \cdots + r^k = \frac{r^{k+1}-1}{r-1}$

(3) $1 + \frac{1}{r} + \frac{1}{r^2} + - \cdots + \frac{1}{r^k} = \frac{1-r^{k+1}}{1-r}$

(4) $1.2 + 2.2^2 + 3.2^3 + - \cdots + k.2^k = (k-1)2^{k+1} + 2$

(5) $1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$

(6) $1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$

2)

a) solve

sol:

$T(n) = 2T(n/2) + \log n$

$T(n) = 2T(n/2) + \log n$

$= 2\left(2T(n/2^2) + \log\frac{n}{2}\right) + \log n = 2^2 T(n/2^2) + 2\log\frac{n}{2} + \log n$

$= 2^2 T(n/2^2) + 2^2 \log\frac{n}{2^2} + 2\log\frac{n}{2} + \log n$

$\vdots$

After K substitutions,

$T(n) = 2^K T(n/2^K) + \log n + 2\log\frac{n}{2} + 2^2\log\frac{n}{2^2} + \cdots + 2^{k-1}\log\frac{n}{2^{k-1}}$

$= 2^K T\left(\frac{n}{2^K}\right) + \log n + 2(\log n - \log 2) + 2^2(\log n - \log 2^2) + \cdots + 2^{k-1}\left(\log n - \log 2^{k-1}\right)$

$= 2^K T\left(\frac{n}{2^K}\right) + \log n + 2\log n - 2\cdot 1 + 2^2\log n - 2\cdot 2^2 + \cdots + 2^{k-1}\log n - (k-1)\cdot 2^{k-1}$

$= 2^K T\left(\frac{n}{2^K}\right) + \log n\left(1 + 2 + 2^2 + \cdots + 2^{k-1}\right) - \left(1\cdot 2 + 2\cdot 2^2 + \cdots + (k-1)2^{k-1}\right)$

$T(n) = 2^K T\left(\frac{n}{2^K}\right) + \log n\left(2^k - 1\right) - \left((k-2)2^k + 2\right)$

Assuming $n = 2^K \Rightarrow K = \log_2 n$

$\Rightarrow T(n) = nT(1) + \log n(n-1) - \left((\log n - 2)n + 2\right)$

$= nc + n\log n - \log n - n\log n + 2n - 2$

$T(n) = cn + 2n - \log n - 2$

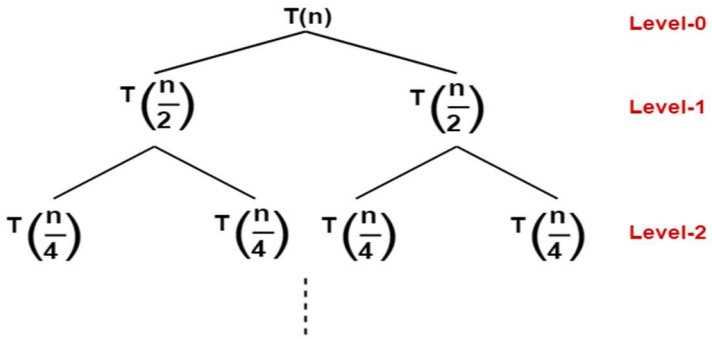
$\Rightarrow T(n) = O(n)$

Solving Recurrence Relations Using Recursion Tree Method: -
(Reference: https://www.gatevidyalay.com/recursion-tree-solving-recurrence-relations/)

Problem-1:
Solve the following recurrence relation using recursion tree method:
$T(n) = 2T(n/2) + n$
Solution-
Step-01:
Draw a recursion tree based on the given recurrence relation.
The given recurrence relation shows-

- A problem of size n will get divided into 2 sub-problems of size n/2.
- Then, each sub-problem of size n/2 will get divided into 2 sub-problems of size n/4 and so on.
- At the bottom most level, the size of sub-problems will reduce to 1.

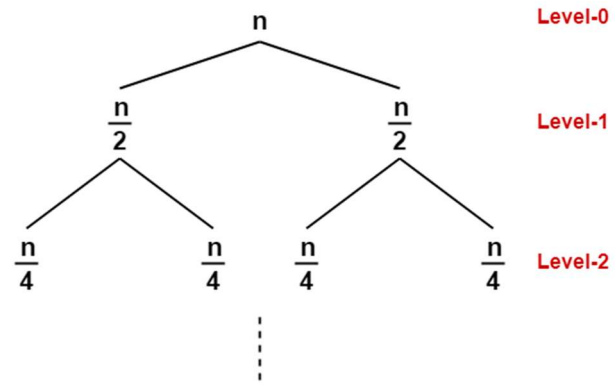
This is illustrated through following recursion tree-



The given recurrence relation shows-

- The cost of dividing a problem of size n into its two sub-problems and then combining their solutions is n.
- The cost of dividing a problem of size n/2 into its two sub-problems and then combining their solutions is n/2 and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem:



Step-02:
Determine cost of each level:

- Cost of level-0 = n
- Cost of level-1 = n/2 + n/2 = n
- Cost of level-2 = n/4 + n/4 + n/4 + n/4 = n, and so on.

Step-03:
Determine total number of levels in the recursion tree:

- Size of problem at level-0 = $n/2^0$
- Size of each sub-problem at level-1 = $n/2^1$
- Size of each sub-problem at level-2 = $n/2^2$

Continuing in similar manner, we have:
Size of each sub-problem at level-i = $n/2^i$
Suppose at level-x (last level), size of each sub-problem becomes 1. Then
$n / 2^x = 1$
$2^x = n$
Taking log on both sides, we get,
$x = \log_2 n$

∴ Total number of levels in the recursion tree = $x+1 = \log_2 n + 1$

Step-04:
Determine number of nodes in the last level:

- Level-0 has 2^0 nodes i.e., 1 node
- Level-1 has 2^1 nodes i.e., 2 nodes
- Level-2 has 2^2 nodes i.e., 4 nodes

Continuing in similar manner, we have,
Last level (i.e., Level - $\log_2 n$) has $2^{\log_2 n} = n^{\log_2 2} = n$ nodes.

Step-05:
Determine cost of last level:
Cost of last level = n * T(1) = cn = θ(n).

Step-06:
Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation:

$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_2 n \text{ levels}} + \theta(n)$$

$= n\log_2 n + \theta(n)$

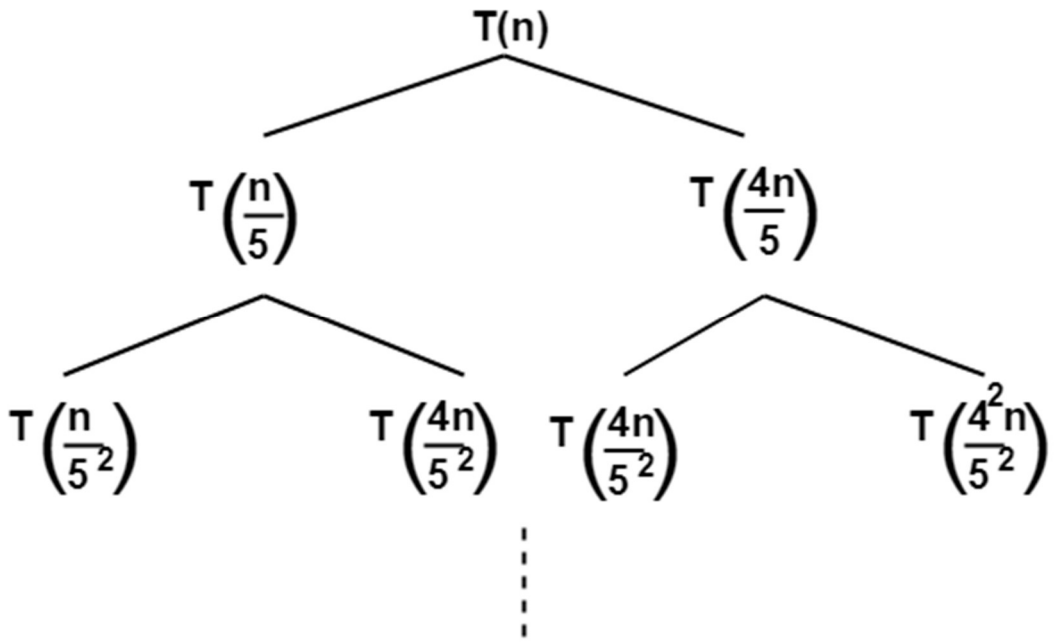
= θ (nlogn)

Problem-2:
Solve the following recurrence relation using recursion tree method:
$T(n) = T(n/5) + T(4n/5) + n$
Solution-

Step-01:
Draw a recursion tree based on the given recurrence relation:
The given recurrence relation shows-

- A problem of size n will get divided into two sub-problems- one of size n/5 and another of size 4n/5.
- Then, sub-problem of size n/5 will get divided into two sub-problems- one of size $n/5^2$ and another of size $4n/5^2$.
- On the other side, sub-problem of size 4n/5 will get divided into two sub-problems- one of size $4n/5^2$ and another of size $4^2n/5^2$, and so on.
- At the bottom most level, the size of sub-problems will reduce to 1.

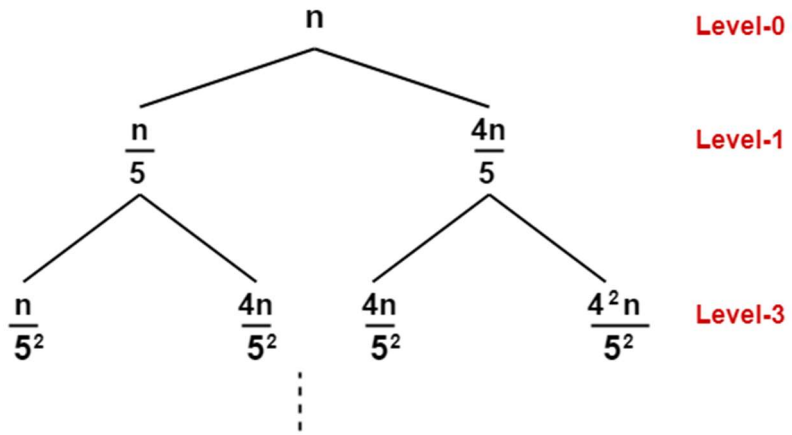
This is illustrated through following recursion tree:



The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/5 into its 2 sub-problems and then combining its solution is n/5.
- The cost of dividing a problem of size 4n/5 into its 2 sub-problems and then combining its solution is 4n/5 and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem:



Step-02:

Determine cost of each level-

- Cost of level-0 = n
- Cost of level-1 = $n/5 + 4n/5 = n$
- Cost of level-2 = $n/5^2 + 4n/5^2 + 4n/5^2 + 4^2n/5^2 = n$

Step-03:

Determine total number of levels in the recursion tree:
We will consider the rightmost subtree (i.e., largest subproblem) as it goes down to the deepest level,

- Size of problem at level-0 = $(4/5)^0 n$
- Size of larger sub-problem at level-1 = $(4/5)^1 n$
- Size of larger sub-problem at level-2 = $(4/5)^2 n$

Continuing in similar manner, we have,
Size of larger sub-problem at level-i = $(4/5)^i n$
Suppose at level-x (last level), size of larger sub-problem becomes 1. Then,
$(4/5)^x n = 1$
$n = (5/4)^x$
Taking log on both sides, we get,
$x = \log_{5/4} n$

∴ Total number of levels in the recursion tree = $x+1 = \log_{5/4} n + 1$

Step-04:

Determine number of nodes in the last level:

- Level-0 has 2^0 nodes i.e., 1 node
- Level-1 has 2^1 nodes i.e., 2 nodes
- Level-2 has 2^2 nodes i.e., 4 nodes

Continuing in similar manner, we have,
Last level (i.e., Level-x=Level - $\log_{5/4}n$) has $2^{\log_{5/4}n}$ nodes = $n^{\log_{5/4}2}$.
Cost of each node in the last level is T(1).

Step-05:
Determine cost of last level:
Cost of last level = $n^{\log_{5/4}2} * T(1) = \theta(n^{\log_{5/4}2})$
Step-06:

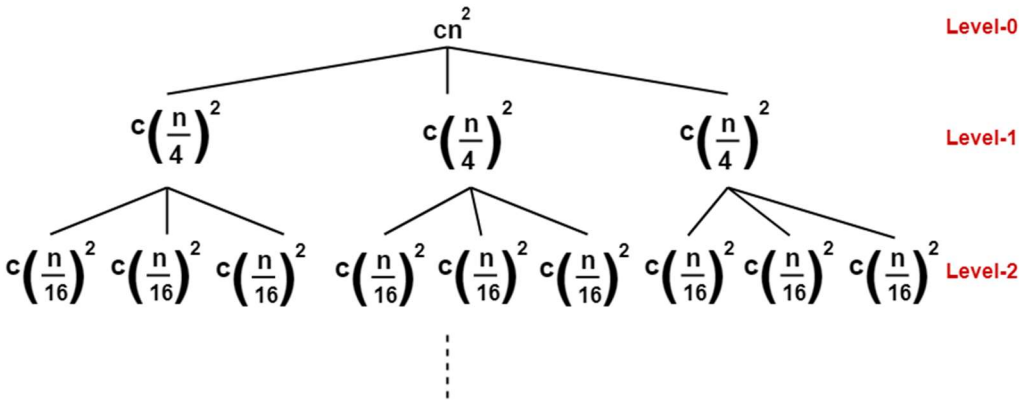
Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation:

$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_{5/4}n \text{ levels}} + \theta(n^{\log_{5/4}2})$$

$= n\log_{5/4}n + \theta(n^{\log_{5/4}2})$
$= \mathbf{\theta(n\log_{5/4}n).}$

Problem-3:
Solve the following recurrence relation using recursion tree method:
$T(n) = 3T(n/4) + cn^2$
Solution-

Step-01:
Draw a recursion tree based on the given recurrence relation:



(Here, we have directly drawn a recursion tree representing the cost of sub problems)

Step-02:

Determine cost of each level:

- Cost of level-0 = cn^2
- Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
- Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$

Step-03:

Determine total number of levels in the recursion tree:

- Size of problem at level-0 = $n/4^0$
- Size of sub-problem at level-1 = $n/4^1$
- Size of sub-problem at level-2 = $n/4^2$

Continuing in similar manner, we have,
Size of sub-problem at level-i = $n/4^i$
Suppose at level-x (last level), size of sub-problem becomes 1. Then,
$n/4^x = 1$
$4^x = n$
$x = \log_4 n$
∴ Total number of levels in the recursion tree = $\log_4 n + 1$

Step-04:

Determine number of nodes in the last level:

- Level-0 has 3^0 nodes i.e., 1 node
- Level-1 has 3^1 nodes i.e., 3 nodes
- Level-2 has 3^2 nodes i.e., 9 nodes

Continuing in similar manner, we have,
Last level (i.e., level - $\log_4 n$) has $3^{\log_4 n} = n^{\log_4 3}$ nodes.

Step-05:

Determine cost of last level:
Cost of last level = $n^{\log_4 3} * T(1) = \theta(n^{\log_4 3})$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation:

$$T(n) = \underbrace{\left\{ cn^2 + \frac{3}{16} cn^2 + \frac{9}{(16)^2} cn^2 + \ldots\ldots\ldots \right\}}_{\text{For } \log_4 n \text{ levels}} + \theta\left(n^{\log_4 3}\right)$$

$= cn^2 \{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \} + \theta(n^{\log_4 3})$

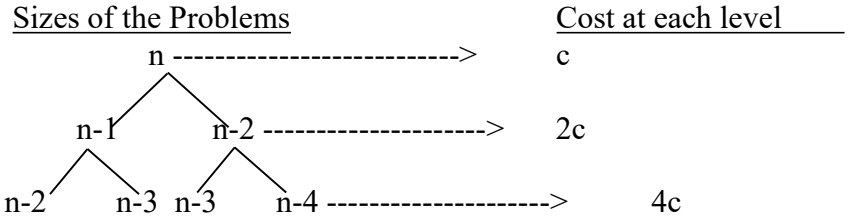
Now, $\{ 1 + (3/16) + (3/16)^2 + \ldots + (3/16)^{i-1} \}$ forms Geometric progression, where
$i = \log_4 n$.

On solving, we get,
$= (16/13)cn^2 \ (1 - (3/16)^{\log_4 n}) + \theta(n^{\log_4 3})$
$= (16/13)cn^2 - (16/13)cn^2 \ (3/16)^{\log_4 n} + \theta(n^{\log_4 3})$
$= (16/13)cn^2 - (16/13)cn^2 \ ((3)^{\log_4 n} / (16)^{\log_4 n}) + \theta(n^{\log_4 3})$
$= (16/13)cn^2 - (16/13)cn^2 \ (n^{\log_4 3} / n^{\log_4 16}) + \theta(n^{\log_4 3})$
$= (16/13)cn^2 - (16/13)cn^2 \ (n^{\log_4 3} / n^2) + \theta(n^{\log_4 3})$
$= (16/13)cn^2 - (16/13)c \ (n^{\log_4 3}) + \theta(n^{\log_4 3})$
$= \mathbf{O(n^2)}$

Problem-4:

T(n)=T(n-1) +T(n-2) + c

Solution:



The size of larger subproblem at last level (level x) is n-x=1. So, x=n-1.
The last level has 2^x nodes. So, the cost of last level is $2^x c$.
The total cost of all levels = $(c+2c+2^2c+\ldots+2^{x-1}c)+2^x c$
$= c(1+2+2^2+\ldots+2^x)$
$= c(2^{x+1}-1)$
$= O(2^{x+1}) = O(2^n)$.

Master Theorem for solving recurrence relations of divide–and-conquer: -

Let $T(n) = aT(n/b)+f(n)$, then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$, for some constant $\epsilon > 0$ *(i.e.,* if $f(n) < n^{\log_b a}$*)*, then $T(n) = \theta(n^{\log_b a})$.

2. (a) If $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a} \lg n)$.
   (b) If $f(n) = \theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$, then $T(n) = \theta(n^{\log_b a} \lg^{k+1} n)$.

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ *(i.e.,* if $f(n) > n^{\log_b a}$*)*, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \theta(f(n))$.

Examples:

(1).

$T(n) = 9T(n/3) + n$.

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

(2).

$T(n) = T(2n/3) + 1,$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2a applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

(3)

$T(n) = 3T(n/4) + n \lg n$,

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

(4)

$T(n) = 2T(n/2) + n \lg n$

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ.

Apply case 2b of master theorem to solve the above problem.
T(n) = $\theta(nlg^2 n)$.

Binary search: -

→ Let a[1:n] be a list of elements that are sorted in ascending order.
→ We have to determine whether a given element x is present in the list or not.
→ If x is present, we have to return its position, else return 0.

Iterative algorithm: -

```
Algorithm BinSearch(a,n,x)
// Given an array a[l:n] of elements in nondecreasing order, n>0, determine
// whether x is present, and if so, return j such that x =a[j]; else return 0.
{
    low := 1; high:=n;
    while (low ≤high) do  //If there is at least one element in the given part of array
    {
                mid: = ⌊(low + high)/2⌋;
                if (x <a[mid]) then high:=mid-1;
                else if (x > a[mid]) then low:= mid+1;
                else return mid;
     }
return 0;
}
```

Recursive algorithm: -

```
Algorithm RBinSearch(a, low, high, x)
{
     // Given an array a[low: high] of elements in nondecreasing
     // order, 1<low<high, determine whether x is present, and
     //if so, return j such that x =a[j]; else return0.

    if (low>high) then //If there are no elements in the given part of array
            return 0;
    else
    {
        // Reduce the problem into a smaller subproblem.
        mid:= ⌊(low + high)/2⌋;
        if (x = a[mid]) then
                return mid;
        else if (x <a[mid]) then
                return RBinSearch(a, low, mid-1, x);
        else
                return RBinSearch(a, mid+1, high, x);
     }
  }
```

→ This algorithm is initially invoked as RBinsearch(a,1, n, x).

Time Complexity:

→ The main problem is divided into one sub-problem in constant time.

→ The answer to the new sub-problem is also the answer to the original problem.
So, there is no need for combining.

→ The running-time of this algorithm can be characterized as follows:

$$T(n) = \begin{cases} c, & if\ n = 1 \\ T(n/2) + c, & otherwise \end{cases}$$

Solving by using substitution method,

T(n) = T(n/2) +c

= [T(n/4) +c] +c = T(n/4) +2c

= [T(n/8) +c] +2c = T(n/8) +3c =T(n/2^3)+3c

After applying this substitution *k* times,

T(n) = T(n/2^k) +kc

To terminate this substitution process, we switch to the closed form *T(1)=c*, which happens when $2^k = n$ which implies $k=log_2 n$

So, *T(n) = T(1) +*$clog_2 n$

T(n) = c +$clog_2 n$

= O(logn) --- Worst-case time complexity

The best-case time complexity is O(1).

Merge sort: -

→ Given a sequence of n elements a[1:n], we divide the given set of elements into two subsets a[1:n/2] and a[n/2+1:n].

→ Each subset is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

Recursive algorithm for merge sort:

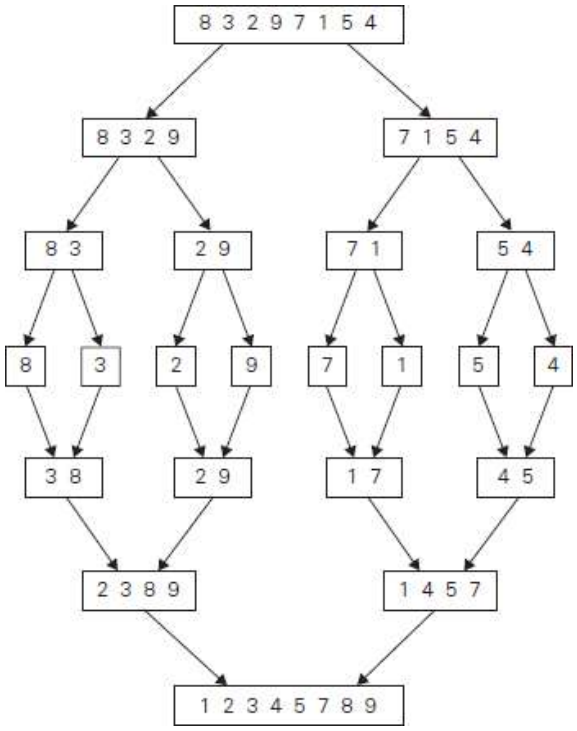
```
Algorithm MergeSort(low, high)
{
   // a[low : high] is a global array to be sorted.

    if (low<high) then  //If there are more than one element in the given part of array
    {
        mid:= ⌊(low + high)/2⌋;
        MergeSort(low, mid);                    // Solve sub-problem1
        MergeSort(mid+1, high);       //Solve sub-problem2
```

```
        Merge(low, mid, high);                    //Combine the solutions of sub-problems
    }
}

Algorithm Merge(low, mid, high)
{
  // a[low :high] is a global array containing two sorted subsets in a[low :mid] and
  // in a[mid+1: high]. The goal is to merge these two sets into a single set residing
  // in a[low :high].
  // b[ ] is an auxiliary(temporary) global array.

      i:=low;
      h: =low;              // h is the indexing variable for the first part of the array a[low: high].
      j: =mid+1;          // j is the indexing variable for the second part of the array a[low :high].
      while ((h≤mid) and (j≤high)) do
      {
         if (a[h]≤a[j]) then
         {
            b[i] :=a[h];  // copy the elements of a into b.
            h:= h+1;
         }
         else
         {
              b[i]:= a[j]; // copy the elements of a into b.
              j:= j+1;
         }
         i:=i+1;
      }  //end of while
      if (h>mid) then
      {
         for k := j to high do
         {
              b[i]:=a[k]; // copy the remaining elements of second part of a into b.
               i:=i+1;
         }
      }
      else
      {
         for k:=h to mid do
         {
               b[i] := a[k]; // copy the remaining elements of first part of a into b.
              i:=i+1;
```

```
        }
    }
    for k: = low to high do
        a[k]:=b[k];   // copy the elements of b into a.
} // end of algorithm.
```

→ MergeSort algorithm is initially invoked as MergeSort(1, n).

Example: Trace the MergeSort on the data: 8,3,2,9,7,1,5,4.



Time Complexity:

→ The merge-sort problem of size *n* is divided into two sub-problems of size *n/2* each.

This division is done in constant time by the statement *mid:*= $\lfloor (low + high)/2 \rfloor$;

→ The solutions to the sub-problems are merged in linear time.

So,

$$T(n) = \begin{cases} c, & if\ n = 1 \\ 2T(n/2) + cn, & otherwise \end{cases}$$

Solving by using substitution method,

$T(n) = 2T(n/2) + cn$

$= 2[2T(n/4) + cn/2] + cn = 4T(n/4) + cn + cn = 4T(n/4) + 2cn$

$= 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn = 2^3 T(n/2^3) + 3c$

After applying this substitution k times,

$T(n) = 2^k T(n/2^k) + kcn$

To terminate this substitution process, we switch to the closed form $T(1)=c$, which happens when $2^k = n$ which implies $k=log_2 n$.

So,

$T(n) = nT(1) + c.\ log_2 n.\ n$

$T(n) = cn + cnlog_2 n$

$T(n) = O(nlogn)$

This is the best-case, worst-case and average-case time complexity.

Quick sort: -

→ In quicksort, the division of a[1:n] into two subarrays is made so that, the sorted sub-arrays do not need to be merged later.

→ This is accomplished by picking some element of a[ ], say *t*, which is called *pivot(or, portioning element)*, and then reordering the other elements such that all the elements which are less than or equal to *t* are placed before *t* and all the elements which are greater than *t* are placed after *t* in the array a[1:n].

This rearrangement is referred to as *partitioning.*

→ Although there are many choices for the pivot, here we assume that the *first element in the array acts as pivot*, for convenience.

→ Partition of a set of elements S about the pivot *t* produces two disjoint subsets S_1 and S_2, where

$S_1 = \{x \in S - \{t\} \mid x \leq t\}$ and

$S_2 = \{x \in S - \{t\} \mid x > t\}$.

→ Obviously, after a partition is achieved, pivot element *t* will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of *t* independently.

→ **Note:** The difference between mergesort and quicksort is as follows:

In mergesort, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; but in quicksort , the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Recursive algorithm for quick sort:

Algorithm QuickSort(low, high)

{

// Sorts the elements a[low], ..., a[high] which reside in the global

// array a[1 :n] into ascending order.

// a[n+1] is considered which must be greater than or equal to all elements in a[1:n].

```
    if (low<high)  then     //If there are more than one element
    {
        j:= Partition(a,low,high+1); // j is the final position of the partitioning element.
        QuickSort(low, j - 1); // Solve subproblem1
        QuickSort(j+1, high); //Solve subproblem2
        // There is no need for combining solutions.
    }
}

Algorithm Partition(a, l, h)
{
    //a[l] is considered pivot
    t:=a[l];
    lp:=l+1;
    rp:=h-1;
    while(lp≤rp) do
    {
        while(a[lp]≤t) do lp:=lp+1;
        while(a[rp]>t) do rp:=rp-1;
        if(lp<rp) then
        {   // swap a[lp] and a[rp]
            temp:= a[lp];
            a[lp]:=a[rp];
            a[rp]:=temp;
        }
    }
    a[l]:=a[rp];
    a[rp]:=t;
    return rp;
}
```

→Initially, QuickSort is invoked as QuickSort(1,n).

Example: Trace the QuickSort algorithm on the following data:
65, 70, 75, 80, 60, 55, 50, 45.

Index:	1	2	3	4	5	6	7	8	9
		lp						*rp*	
Elements:	**65**	70	75	80	60	55	50	45	∞
		lp						*rp*	
	65	45	75	80	60	55	50	70	∞

		lp				*rp*		
65	45	75	80	60	55	50	70	∞
		lp				*rp*		
65	45	50	80	60	55	75	70	∞
			lp		*rp*			
65	45	50	80	60	55	75	70	∞
			lp		*rp*			
65	45	50	55	60	80	75	70	∞
				rp	*lp*			
65	45	50	55	60	80	75	70	∞
60	45	50	55	**65**	80	75	70	∞
	lp		*rp*					
60	45	50	55					∞
			rp	*lp*				
60	45	50	55					∞
55	45	50	**60**					∞
	lp	*rp*						
55	45	50						∞
		rp	*lp*					
55	45	50						∞
50	45	**55**						∞
50	*lp, rp*							
	45							∞
	rp	*lp*						
50	45							∞
45	**50**							∞
						lp	*rp*	
					80	75	70	∞
							rp	*lp*
					80	75	70	∞
					70	75	**80**	∞
						lp, rp		
					70	75		∞
					rp	*lp*		
					70	75		∞

70 **75** ∞

45 **50** **55** **60** **65** **70** **75** **80**

So, the sorted data is: 45,50, 55, 60, 65, 70, 75, 80.

Time Complexity:

Quick Sort partitions the given set of elements S into two subsets S1 and S2 around pivot element *t*. That means, S=S1 U {*t*} U S2.

If |S| = n and |S1| = i, then |S2| = n-i-1. So, $T(n) = T(i) + T(n-i-1) + f(n)$

$T(n) = T(i) + T(n-i-1) + cn$; since the partitioning is done in linear time.

(i) Worst-Case Analysis:

This occurs when the elements are in either ascending order or descending order.

→Consider that the elements are in ascending order. Then,

|S1| = 0 and |S2| = n-1.

So,

$T(n) = T(0) + T(n-1) + cn$

$T(n) = c + T(n-1) + cn$

Neglecting the insignificant constant term,

$T(n) = T(n-1) + cn$

Solving using substitution method,

$T(n) = T(n-2) + cn + c(n-1)$

$T(n) = T(n-3) + cn + c(n-1) + c(n-2)$

After k substitutions,

$T(n) = T(n-k) + cn + c(n-1) + c(n-2) + \ldots + c(n-(k-1))$

$= T(n-k) + c(n+(n-1)+(n-2)+\ldots+(n-(k-1))$

Assume n=k,

$T(n) = T(0) + c(k+(k-1)+(k-2)+\ldots+1)$

$= c + c(1+2+\ldots+k)$

$= c + c(k(k+1)/2)$

$= c + c(n(n+1)/2)$ $T(n) = O(n^2)$.

(ii) Best-case Analysis:

This occurs when the pivot element occupies middle position after partitioning. In this case, |S1| = n/2 and |S2| = n/2.

So,

$T(n) = 2T(n/2) + cn$

Solving this using substation method,

$T(n) = 2T(n/2) + cn$
$= 2[2T(n/4) + cn/2] + cn = 4T(n/4) + cn + cn = 4T(n/4) + 2cn$
$= 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn = 2^3T(n/2^3) + 3c$

After applying this substitution *k* times,
$T(n) = 2^kT(n/2^k) + kcn$

To terminate this substitution process, we switch to the closed form $T(1)=c$, which happens when $2^k = n$ which implies $k=log_2 n$.

So,
$T(n) = nT(1) + c.log_2 n.n$
$T(n) = cn + cnlog_2 n$
$T(n) = O(nlogn)$.

(iii) Average-Case Analysis: (Using Probability Analysis)
→After partitioning is performed, the pivot element may occupy any position from 1 to *n* in the array. So, S_1 can have any size from 0 to n-1 with equal probability of 1/n. Similarly, S_2 also can have any size from 0 to n-1 with equal probability of 1/n.

And we know that $T(n) = T(i) + T(n-i-1) + cn$.

→So, the expected or average value of $T(i)$ is $\frac{\sum_{j=0}^{n-1} T(j)}{n}$.

Similarly, the expected or average value of $T(n-i-1)$ is also $\frac{\sum_{j=0}^{n-1} T(j)}{n}$.

Now, the equation $T(n) = T(i) + T(n-i-1) + cn$, becomes

$$T(n) = \frac{2\sum_{j=0}^{n-1} T(j)}{n} + cn \quad \text{--------} \quad (1)$$

If equation (1) is multiplied by n, it becomes

$$nT(n) = 2\sum_{j=0}^{n-1} T(j) + cn^2 \quad \text{--------} \quad (2)$$

From (2) we can get

$$(n-1)T(n-1) = 2\sum_{j=0}^{n-2} T(j) + c(n-1)^2 \quad \text{--------} \quad (3)$$

Subtracting (3) from (2),

$$nT(n)-(n-1)T(n-1) = 2T(n-1) + c(n^2-(n-1)^2)$$

$nT(n) = (n+1)\ T(n-1) + 2nc\text{-}c$

Dropping insignificant term –c, we get
$nT(n) = (n+1)\ T(n-1) + 2nc$ -------- (4)

Dividing (4) by $n(n+1)$, we get
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \quad \text{-------- (5)}$$

Solving the above equation using substitution method,
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n)}{n+1} = \left(\frac{T(n-2)}{n-1} + \frac{2c}{n}\right) + \frac{2c}{n+1}$$

$$\frac{T(n)}{n+1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1}$$

After k substitutions,
$$\frac{T(n)}{n+1} = \frac{T(n-k)}{n-k+1} + 2c\left(\frac{1}{n-k+2} + \frac{1}{n-k+3} + \ldots + \frac{1}{n+1}\right)$$

When n=k,
$$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1}\right)$$

$$\frac{T(n)}{n+1} = T(0) + 2c\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1} - 1\right)$$

Since $\sum_{i=1}^{n} \frac{1}{i} \approx \log_e n + \gamma$, where $\gamma = 0.577$ (Euler's constant),

$$\frac{T(n)}{n+1} = c + 2c(\log_e(n+1) + \gamma - 1)$$
$$= c + 2c\log_e(n+1) + 2c(\gamma - 1)$$
$$= c + 2c\log_e(n+1)$$
$$T(n) = (n+1)(c + 2c\log_e(n+1))$$
$$= c(n+1) + 2c(n+1)\log_e(n+1)$$
$$= c(n+1) + 2cn\log_e(n+1) + 2c\log_e(n+1)$$
$$T(n) = O(n\log n)$$

THE GREEDY METHOD

→The Greedy method is used to solve many of the optimization (i.e., minimization or maximization) problems.

→Some optimization problems also involve some constraints. A solution that satisfies these constraints is called feasible solution. A feasible solution which optimizes (i.e., minimizes or maximizes) a given objective function is called optimal solution.

→Before Greedy algorithm begins, we set up a selection criterion (called greedy criterion) which decides the order of selection of inputs.

→ The greedy approach suggests constructing a solution through a sequence of steps, each step expanding a partially constructed solution obtained so far until a complete solution to the problem is reached.

→On each step (and this is the central point of this technique) the choice made must be:

- *feasible*, i.e., it has to satisfy the problem's constraints.
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step.
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm.

→These requirements explain the technique's name: at each step, it suggests a "greedy" grab of the best alternative available, in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.

→There are problems for which a sequence of locally optimal choices yields an optimal solution for every instance of the given problem.

→However, there are some other problems for which this is not the case (that means every time we may not get optimal solution). For such problems, a greedy algorithm can still be of value if we are interested in an approximate (or, near optimal) solution.

CONTROL ABSTRACTION FOR GREEDY METHOD:

```
Algorithm Greedy(a,n)
// a[l : n] contains the n inputs.
{
        solution:=Φ ;       // Initialize the solution.
        for i:= 1 to n do
        {
             x := Select(a); // Select next i/p from 'a' based on greedy
                             // criterion and assign its value to x
            if (Feasible(solution, x)) then
            {
            // Check whether the inclusion of x into partially constructed solution results in
            //  feasible solution
                    solution := Union(solution, x) ;
             }
        }
 return solution;
```

FRACTIONAL KNAPSACK PROBLEM:

→We are given a set of 'n' items (or, objects), such that each item i has a weight 'w_i' and a profit 'p_i'. We wish to pack a knapsack whose capacity is 'M' with a subset of items such that total profit is maximized.

→Further, we are allowed to break each item into fractions arbitrarily. That means, for an item i we can take an amount $w_i x_i$ (which gives a profit of $p_i x_i$) such that $0 \le x_i \le 1$, and $\sum_{i=1}^{n} w_i x_i \le M$

→Formally the fractional knapsack problem can be stated as follows:

Maximize $\sum_{i=1}^{n} p_i x_i$

Subject to the following constraints:

$\sum_{i=1}^{n} w_i x_i \le M$

and

$0 \le x_i \le 1, (1 \le i \le n)$.

→The solution to this problem is represented as the vector (x_1, x_2, …, x_n).

Note: If the sum o f all weights is less than or equal to M (i.e., $\sum_{i=1}^{n} w_i x_i \leq M$), then all the items can be placed in the knapsack which results in the solution x_i=1, 1 ≤ i ≤ n.

POSSIBLE GREEDY STRATEGIES:

There are several greedy strategies possible. In each of these strategies the knapsack is packed in several stages.

In each stage, one item is selected for inclusion into the knapsack using the chosen strategy.

1. TO BE GREEDY ON PROFIT: (Don't use it)

The selection criterion is "From the remaining objects, select the object with maximum profit that fits into the knapsack".

Using this criterion, the object with the largest profit is packed first (provided enough capacity is available), then the one with next largest, and so on.

That means the objects are selected in the decreasing order of their profit values.

This strategy does not always guarantee an optimal solution, but only a suboptimal solution.

Example: Consider the following instance of the knapsack problem:
n = 3, M = 20, (p_1, p_2, p_3) = (24, 25, 15), and (w_1, w_2, w_3) = (15, 18, 10).
Solution: Objects are selected and placed into the knapsack in decreasing order of their profits.

i	1	2	3
p_i	24	25	15
w_i	15	18	10
Order of Selection (or) Rank)	2	1	3
x_i	2/15	1	0

So when we are greedy on profit, we obtain the solution as
$(x_1, x_2, x_3) = (\frac{2}{15}, 1, 0)$
which gives a profit of $(24\times\frac{2}{15}) + (25\times1) + (15\times0) = 3.2 + 25 + 0 = 28.2$

It is not an optimal solution, as there is a superior solution to this, i.e.,
$(x_1, x_2, x_3) = (1, 0, \frac{1}{2})$ which gives a profit of 24+15*/2 = 24 + 7.5 = 31.5.

2. <u>TO BE GREEDY ON WEIGHT:</u> (Don't use it)

The selection criterion is "From the remaining objects select the one with minimum weight that fits into knapsack."

That means, the objects are selected in increasing order of theirweights. It does not always yield optimal solution.

<u>Example:</u> Consider the following instance of the knapsack problem:
n = 3, M = 20, (p_1, p_2, p_3) = (24, 25, 15), and (w_1, w_2, w_3) = (15, 18, 10).
<u>Solution:</u> Objects are selected and placed into the knapsack in increasing order of their weights.

i	1	2	3
p_i	24	25	15
w_i	15	18	10
Order of Selection (or) Rank)	2	3	1
x_i	$\frac{10}{15} = \frac{2}{3}$	0	1

So when we are greedy on weights, we obtain the solution as

$(x_1, x_2, x_3) = (\frac{2}{3}, 0, 1)$

which yields a profit of $(24 \times \frac{2}{3}) + (25x0) + (15 \times 1) = 16 + 0 + 15 = 31$.

It is not an optimal solution, as there is a superior solution to this, i.e.,
$(x_1, x_2, x_3) = (1, 0, \frac{1}{2})$ which gives a profit of 24+15*/2 = 24 + 7.5 = 31.5.

3. <u>TO BE GREEDY ON PROFIT DENSITY (PROFIT PER UNIT WEIGHT):</u>

The selection criterion is "From the remaining objects, select the one with maximum p_i/w_i ratio that fits into the knapsack."

That means, the objects are selected in decreasing order of their p_i/w_i ratios.

This strategy always produces an optimal solution to the fractional knapsack problem.

<u>Example:</u> Consider the following instance of the knapsack problem:
n = 3, M = 20, (p_1, p_2, p_3) = (24, 25, 15), and (w_1, w_2, w_3) = (15, 18, 10).

Solution:

i	1	2	3
p_i	24	25	15
w_i	15	18	10
p_i/w_i	$\frac{24}{15} = 1.6$	$\frac{25}{18} = 1.38$	$\frac{15}{10} = 1.5$
Order of Selection (or) Rank)	1	3	2
x_i	1	0	$\frac{5}{10} = \frac{1}{2}$

The optimal solution is: $(x_1, x_2, x_3) = (1, 0, \frac{1}{2})$

which yields a profit of $24*1 + 25*0 + 15*\frac{1}{2} = 24 + 0 + 7.5 = 31.5$.

Algorithm for knapsack problem using greedy method:

```
Algorithm Greedy_knapsack (M, n)
// p [l: n] and w [1: n] contain the profits and weights respectively of n
//objects which are ordered such that p[i] / w[i] ≥ p[i+1] / w[i+1]
// M is the knapsack capacity and x[1:n] is the solution vector.
{
        for i := 1 to n do
            x[i] := 0.0;        //Initialization of solution vector
        RC := M; // Remaining capacity knapsack
        for i := 1 to n do
        {
            if (w[i] > RC) then break;
            else
            {
                x[i] = 1.0;
                RC= RC- w[i];
            }
        }
        if (i≤n) then x[i] = RC/w[i] ;
}
```

Ignoring the time taken initially to sort the objects according to their profit densities, the time

complexity of this algorithm O(n).

Exercises:

1) n=7, M = 15, (p1, p2, p3, p4, p5, p6, p7) = (10, 5, 15, 7, 6, 18, 7), and (w1, w2, w3,w4,w5,w6,w7) = (2,3,5,7,1,4,1).

2) n=7, M = 15, (p1,p2,p3,p4,p5,p6,p7) = (10, 5, 15, 7, 6, 18, 3), and (w1,w2,w3,w4,w5,w6,w7) = (2,3,5,7,1,4,1).

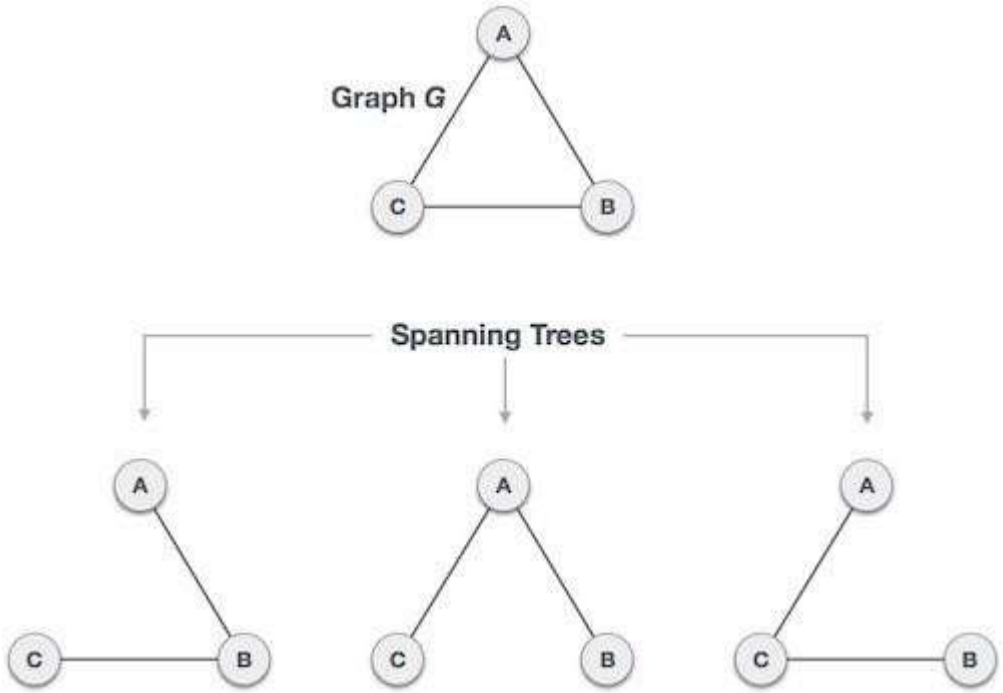
SPANNING TREE:

→Let G = (V, E) be an undirected connected graph. A subgraph G' = (V, E') of G (where E' is subset of E) is called a spanning tree of G iff G' is a tree.

→The spanning tree of a given connected graph with *n* vertices is a connected subgraph with *n* vertices but without any cycles. That means it will have *n*-1 edges.

→Every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree.

Example: A complete graph with three nodes together with all of its spanning trees.



General Properties of Spanning Tree:

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a cycle, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree:

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).

- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

MINIMUM SPANNING TREE (or, MINIMUM COST SPANNING TREE):

→Given a weighted connected graph G, a ***minimum spanning tree (***MST) is its spanning tree with the *smallest sum of the weights on all its edges*.

→The ***minimum cost spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

→There are two algorithms available to find an MST for a given graph:

1. Prims' Algorithm
2. Kriskal's algorithm

PRIM'S ALGORITHM:

→It is a greedy algorithm to obtain a minimal cost spanning tree.

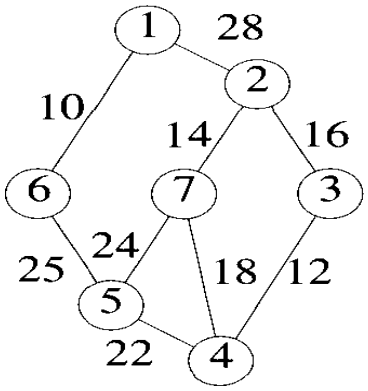
→It builds MST edge by edge.

→The next edge to include is chosen according to the following selection criterion: *"Choose an edge that results in a minimum increase in the sum of the costs of the edges so far included provided that the inclusion of it will form a tree."*

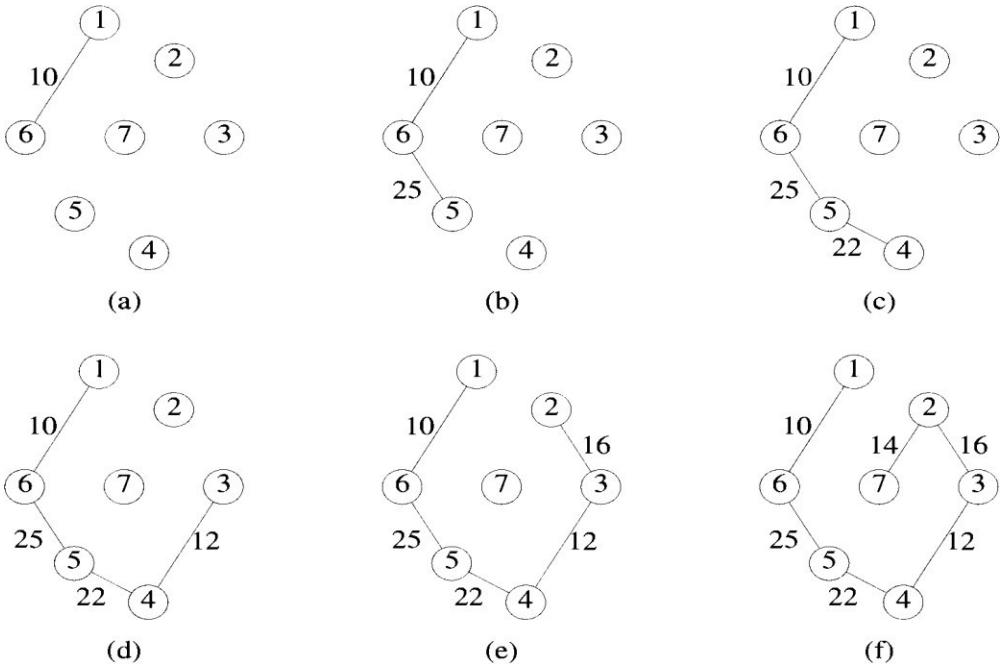
→That means, among the edges which are incident on vertices that are selected so far, select the one with minimum cost, provided that the inclusion of it will not result in a cycle.

→ This algorithm guarantees that a tree is formed at each intermediate stage.

Example: Find the minimum cost spanning tree of the following graph using Prim's algorithm, showing different stages in constructing the tree.

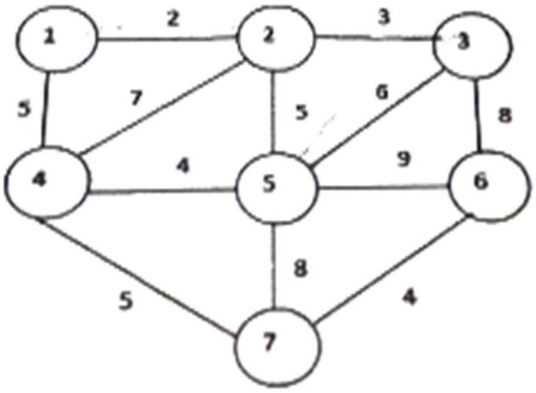


Stages in Prim's algorithm:



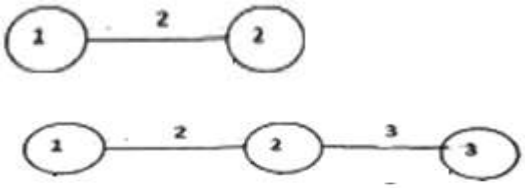
Cost of minimum spanning tree is = 10 + 25 + 22 + 12 + 16 + 14 = 99.

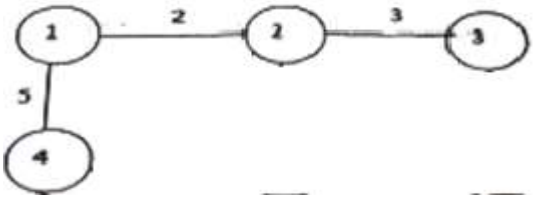
<u>Example:</u> Apply Prim's algorithm on the following graph to find MST.

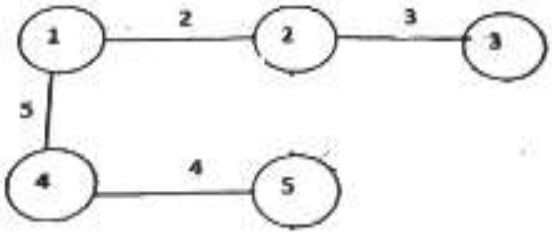


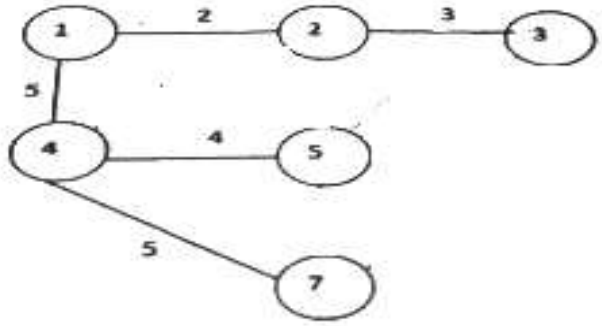
<u>Solution:</u>

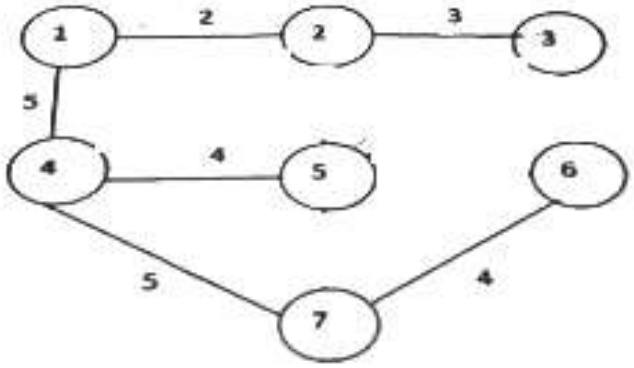
<u>Steps in constructing MST:</u>











The cost of MST = 2+3+5+4+5+4 = 23.

Note: More than one MST is possible but cost must be same.

ALGORITHM:

Algorithm Prim(E, cost, n, t)

// E is the set of edges in G.

// cost[l : n,1 : n] is the cost adjacency matrix of an n vertex graph such that

// cost[i,j] is either a positive real number or ∞ if no edge (i, j) exists.

```
//A minimum spanning tree is computed and stored as a set of edges in the
// array t[1:n-1,1: 2].
// (t[i,1],t[i,2]) is an i^th edge in the minimum-cost spanning tree.
// The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[l, l] := k; t[1, 2] := l;
    for i := 1 to n do       // Initialize near[ ] array.
        if (cost[i,l] < cost[i,k]) then        // If l is nearer to i than k
            near[i] := l;
        else
            near[i] := k;
    near[k] := near[l] := 0;
    for i:= 2 to n -1 do
        {
        // Find n-2 additional edges for t.
            Let j be an index such that near[j] ≠ 0 and cost[j, near [j]] is minimum;
            t [i, 1] := j;
            t[i, 2] := near[j];
            mincost := mincost + cost[j, near [j]];
            near[j] :=0;
            for k :=1 to n do //update near[ ] array
                if (near [k] ≠ 0) and ((cost [k, near [k]) > (cost [k, j])) then
                    near [k] :=j;
         }
    return mincost;
}
```

NOTE: To efficiently determine the next edge (i, j) to be added in the spanning tree, we use near[ ] array.

→The running time of Prim's algorithm is $O(n^2)$.

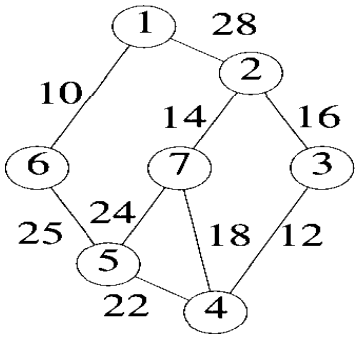
Kruskal's Algorithm:

→It is also a greedy method to obtain a minimal cost spanning tree.
→It also builds MST edge by edge.
→The next edge to include is chosen according to the following selection criterion: *"Choose an edge with smallest weight provided that the inclusion of it will not result in a cycle."*

→In other words, the edges are selected in increasing order of weights provided cycle will not be created at any stage.

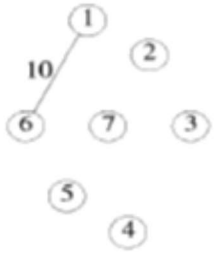
Note: Unlike in Prim's algorithm, the set of edges so far included need not form a tree at all stages in Kruskal's algorithm, i.e., a forest of trees may be generated. But at the end, a tree will be generated.

Example: Find the minimum cost spanning tree of the following graph using Kruskal's algorithm, showing different stages in constructing the tree.

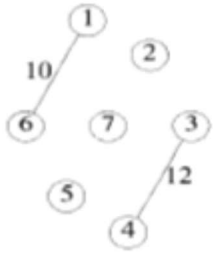


Stages in Kruskal's algorithm:

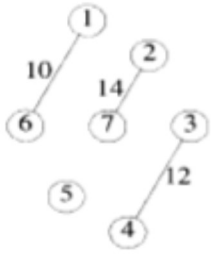
(a)



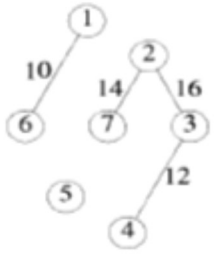
(b)



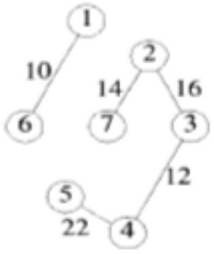
(c)



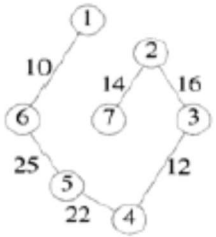
(d)



(e)



(f)



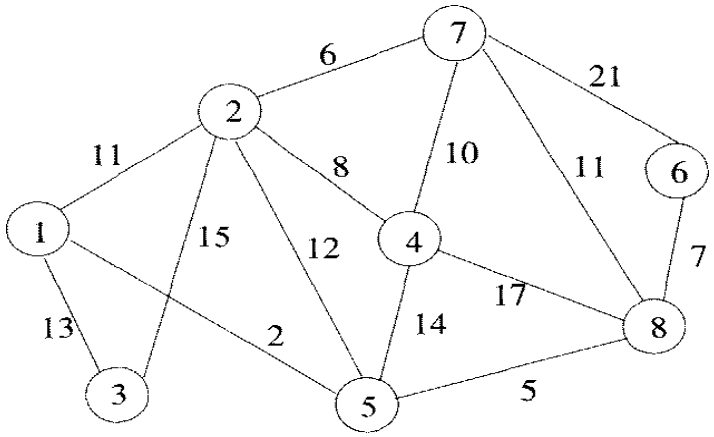
(g)

Cost of minimum spanning tree = 10 + 12 + 14 + 16 + 22 + 25 = 99.

Simple version of Kruskal's algorithm:

```
t = Φ;
while ((t has less than n - 1 edges) and (E ≠ Φ)) do
{
    choose an edge (v, w) of lowest weight from E;
    delete (v, w) from E;
    if (v, w) does not create a cycle in t then
        add (v, w) to t;
    else discard (v, w);
}
```

Exercise: Compute MST for the following graph using Prim's and Kruskal's algorithms:



SINGLE SOURCE SHORTEST PATHS PROBLEM:
(DIJKSTRA'S ALGORITHM)

PROBLEM DESCRIPTION:

We are given a directed graph (digraph) G = (V, E) with the property that each edge has a non-negative weight. We must find the shortest paths from a given source vertex v_0 to all the remaining vertices (called destinations) to which there is a path.

Length of the path is the sum of the weights of the edges on the path.

A GREEDY SOLUTION:

We can solve the shortest paths problem using a greedy algorithm, developed by Dijkstra, that generates the shortest paths in stages.

In each stage, a shortest path to a new destination vertex is generated.

The destination vertex for the next shortest path is selected using the following greedy criterion:

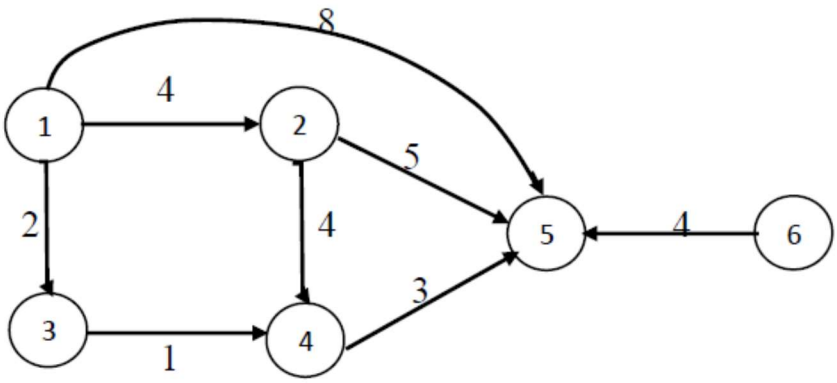
"From the set of vertices, to which a shortest path has not been generated, select one that results in the least path length (or, one that is closest to the source vertex)."

In other words, Dijkstra's method generates the shortest paths in increasing order of path lengths.

→We represent an n vertex graph by an $n \times n$ cost adjacency matrix *cost*, with

cost[i, j] being the weight of the edge <i, j>. In case the edge <i, j> is not present in the graph, its cost *cost[i, j]* is set to some large number (∞). For i = j, *cost[i, j]* can be set to any non-negative number such as 0.

Example:



Cost adjacency matrix:

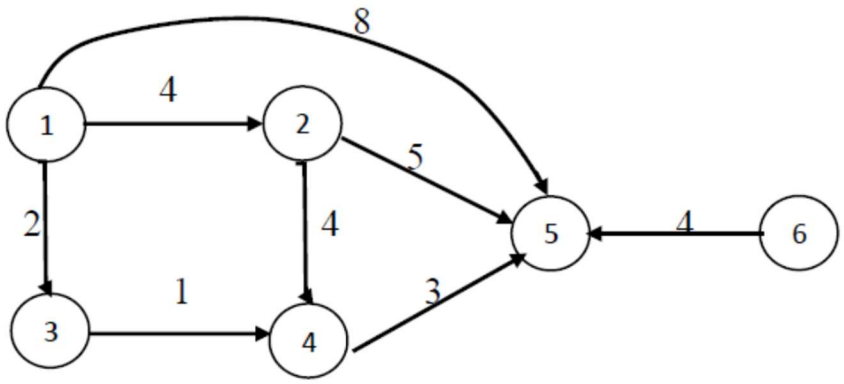
$$\begin{bmatrix} 0 & 4 & 2 & \infty & 8 & \infty \\ \infty & 0 & \infty & 4 & 5 & \infty \\ \infty & \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 4 & 0 \end{bmatrix}$$

→Let 'S' denote the set of vertices (including source vertex v_0) to which the shortest paths have already been generated.

The set 'S' is maintained as a bit array with S[i] = 0 if vertex 'i' is not in 'S' and S[i] = 1 if vertex 'i' is in 'S'.

→ To store the lengths of the resultant shortest paths from the source vertex to remaining vertices, we use an array *dist[1:n]*.

Example: Obtain the shortest paths in increasing order of lengths from vertex 1 to all remaining vertices in the following digraph.



Solution:

Step1: Finding the next shortest path

$1 \xrightarrow{4} 2$

$1 \xrightarrow{2} 3$ (path with minimum distance)

$1 \xrightarrow{8} 5$

$1 \xrightarrow{\infty} 4$

$1 \xrightarrow{\infty} 6$

Step2:

(i) Update distances of those nodes adjacent to 3 (if possible) which are not covered:

$1 \xrightarrow{2} 3 \xrightarrow{1} 4$

(ii) Finding the next shortest path

$1 \xrightarrow{4} 2$

$1 \xrightarrow{2} 3 \xrightarrow{1} 4$ (path with minimum distance)

$1 \xrightarrow{8} 5$

$1 \xrightarrow{\infty} 6$

Step 3:

(i) Update distances of those nodes adjacent to 4 (if possible) which are not covered:

$1 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{3} 5$

(ii) Finding the next shortest path

$1 \xrightarrow{4} 2$ (path with minimum distance)

$1 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{3} 5$

$1 \xrightarrow{\infty} 6$

Step 4:

(i) Update distances of those nodes adjacent to 2 (if possible) which are not covered:

Node 5 is adjacent to node 2. But distance of 5 via node 2 (i.e., 9) will be more than its current distance (i.e., 6). So, its distance is not updated.

(ii) Finding the next shortest path

$1 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{3} 5$ (path with minimum distance)

$1 \xrightarrow{\infty} 6$

Step 5:

(i) Update distances of those nodes adjacent to 5 (if possible) which are not covered:

Nothing is adjacent to node 5.

(ii) Finding the next shortest path

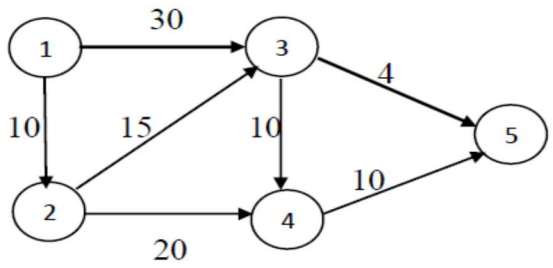
$1 \xrightarrow{\infty} 6$

There is no path to node 6.

Shortest paths in increasing order:

Shotest Paths	**Lengths**
$1 \xrightarrow{2} 3$	2
$1 \xrightarrow{2} 3 \xrightarrow{1} 4$	3
$1 \xrightarrow{4} 2$	4
$1 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{3} 5$	6

Exercise: Obtain the shortest paths in increasing order of lengths from vertex 1 to all remaining vertices in the following digraph.



ALGORITHM FOR SHORTEST PATHS:

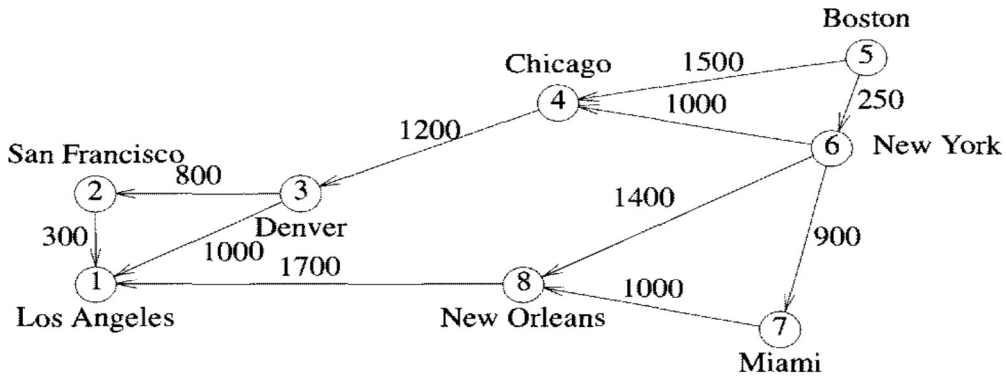
Algorithm ShortestPaths(*v*, cost, dist, n)
//graph G with n vertices is represented by its cost adjacency matrix cost[1:n, 1:n].
//dist[i], 1≤ i ≤ n, is set to the length of the shortest path from source vertex v to the vertex i.

```
{
    for i:= 1 to n do
    {
        S[i] := 0; // Initialize S[ ] array
        dist[i] := cost[v, i];        //Initialize dist[ ] array
    }
    S[v] := 1;      //put v in S
    dist[v] := 0;
    for j := 2 to n do
    {
        //determine n-1 paths from v
        choose a vertex u from among those vertices not in S, such that dist[u] is minimum;
        S[u] := 1;      // put u in S
        for (each vertex w adjacent to u with S[w] = 0) do
        {
            // Update distances.
```

```
            if (dist[w] > (dist[u] +cost[u, w])) then
            {
                dist[w] := dist[u] +cost[u, w];
            }
        }
    }
}
```

Algorithm to find out paths also:

```
Algorithm ShortestPaths(v, cost, dist, predecessor, n)
 //graph G with n vertices is represented by its cost adjacency matrix cost[1:n, 1:n].
//dist[i], 1≤ i ≤ n, is set to the length of the shortest path from source vertex v to the vertex i.
{
    for i:= 1 to n do
    {
        S[i]:= 0;                    // Initialize S[ ] array
        dist[i]:= cost[v, i];        //Initialize dist[ ] array
        if(i is adjacent to v) then
            predecessor[i]:= v;

    }
    S[v]:= 1;                    //put v in S
    predecessor[v] :=0;          // source vertex has no predecessor
    dist[v] := 0;
    for j:= 2 to n do
    {
        //determine n-1 paths from v
        choose a vertex u from among those vertices not in S, such that dist[u] is minimum;
        S[u] := 1;      // put u in S
        for (each vertex w adjacent to u with S[w] = 0) do
        {
            // Update distance to w.
            if (dist[w] > (dist[u] +cost[u, w])) then
            {
                dist[w]:= dist[u] +cost[u, w];
                predecessor[w]:= u;
            }
        }
    }
}
```



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Actions for Finding Shortest Paths from vertex 5:

Iteration	S	Distance to vertices								Vertex Selected	Predecessor for selected vertex	Path	Path Length
		[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]				
1	{5}	∞	∞	∞	1500	0	250	∞	∞	**6**	5	5→6	250
2	{5,6}	∞	∞	∞	1250	-	-	1150	1650	**7**	6	5→6→7	1150
3	{5,6,7}	∞	∞	∞	1250	-	-	-	1650	**4**	6	5→6→4	1250
4	{5,6,7,4}	∞	∞	2450	-	-	-	-	1650	**8**	6	5→6→8	1650
5	{5,6,7,4,8}	3350	∞	2450	-	-	-	-	-	**3**	4	5→6→4→3	2450
6	{5,6,7,4,8,3}	3350	3250	-	-	-	-	-	-	**2**	3	5→6→4→3→2	3250
7	{5,6,7,4,8,3,2}	3350	-	-	-	-	-	-	-	**1**	8	5→6→8→1	3350

UNIT-III

DYNAMIC PROGRAMMING

→ This technique is used to solve optimization problems.
→ In dynamic programming, we obtain the solution to a problem by performing a sequence of decisions. We examine the decision sequence to see, if the optimal decision sequence contains optimal decision subsequences.
→ In dynamic programming, an optimal sequence of decisions is obtained by using the principle of optimality.

PRINCIPLE OF OPTIMALITY:

The principle of optimality states that, "In an optimal sequence of decisions each subsequence must also be optimal".
(OR)
The optimal solution to a problem is composed of optimal solutions to the subproblems.

0/1 KNAPSACK PROBLEM:

→This is similar to fractional knapsack problem, except that x_i's are restricted to have a value either 0 or 1.

→ We are given a set of 'n' items (or, objects), such that each item i has a weight 'w_i' and a profit 'p_i'. We wish to pack a knapsack whose capacity is 'M' with a subset of items such that total profit is maximized.

→ The solution to this problem is expressed as a vector $(x_1, x_2, \ldots, x_n)$,where each x_i is 1 if object i has been placed in knapsack, otherwise x_i is 0.

→ The mathematical formulation of the problem is:

$$\text{maximize } \sum_{i=1}^{n} p_i x_i$$

subject to the constraints:

$$\sum_{i=1}^{n} w_i x_i \leq M$$

and

$$x_i \in \{0,1\},\ 1 \leq i \leq n$$

→ We need to make the decisions on the values of $x_1, x_2, x_3, \ldots, x_n$.

→ When optimal decision sequence contains optimal decision subsequences, we can establish recurrence equations that enable us to solve the problem in an efficient way.

→ We can formulate recurrence equations in two ways:
Forward approach
Backward approach

Recurrence equation for 0/1 knapsack problem using Forward approach:
For the 0/1 knapsack problem, optimal decision sequence is composed of optimal decision subsequences.

Let *f(i, y)* denote the value (or profit), of an optimal solution to the knapsack instance with remaining capacity *y* and remaining objects *i, i+1,..., n* for which decisions have to be taken.

It follows that

$$f(n,y) = \begin{cases} p_n, & if\ w_n \le y \\ 0, & if\ w_n > y \end{cases} \quad \text{-------(1)}$$

and

$$f(i,y) = \begin{cases} max\left(f(i+1,y),\ p_i + f(i+1, y - w_i)\right), & if\ w_i \le y \\ f(i+1,y), & if\ w_i > y \end{cases} \quad \text{--(2)}$$

NOTE: when $w_i > y$, we cannot place the object *i*, and there is no choice to make, but when $w_i \le y$ we have two choices, viz., if object *i* can be placed or not. Here we will consider the choice into account which results in maximum profit.

→ *f(1, M)* is the value (profit) of the optimal solution to the knapsack problem we started with. Equation (2) may be used recursively to determine *f(1, M)*.

Example 1: Let us determine *f(1, M)* recursively for the following 0/1 knapsack instance: $n=3$; $(w_1, w_2, w_3) = (100,14,10)$; $(p_1, p_2, p_3) = (20,18,15)$ and $M=116$.

Solution:

$f(1,116) = max\{f(2, 116),\ 20 + f(2, 116 - 100)\}$, since $w1<116$
$= max\{f(2, 116),\ 20 + f(2, 16)\}$
$f(2,116) = max\{f(3, 116), 18 + f(3, 116 - 14)\}$, since $w2<116$

$= max\{f(3, 116), 18 + f(3, 102)\}$

$f(3,116) = 15$ *(since w3<116)*

$f(3,102) = 15$ *(since w3<102)*

$f(2,116) = max\{15, (18 + 15)\} = max\{15, 33\} = 33$

now,

$f(2, 16) = max\{f(3, 16), 18 + f(3,2)\}$

$f(3,16) = 15$ *(since w3<16)*

$f(3,2) = 0$ *(since w3>2)*

So, $f(2,16) = max(15, 18+0) = 18$

$f(1,116) = max\{33, (20 + 18)\} = 38$

Tracing back the x_i values:

To obtain the values of x_i's, we proceed as follows:

If $f(1, M) = f(2,M)$ then we may set $x_1=0$.

If $f(1, M) \neq f(2,M)$ then we may set $x1=1$. Next, we need to find the optimal solution that uses the remaining capacity $M-w1$. This solution has the value $f(2, M-w1)$. Proceeding in this way, we may determine the values of all the x_i's.

Determining the x_i values for the above example:

$f(1, 116) = 38$

$f(2, 116) = 33$

Since $f(1, 116) \neq f(2, 116) \rightarrow x_1 = 1$.

After placing object *1*, remaining capacity is *116 -100 =16*. This will lead to *f(2, 16)*.

$f(2,16)=18$

$f(3,16)=15$

Since $f(2, 16) \neq f(3, 16) \rightarrow x_2 = 1$.

After placing object *2*, remaining capacity is *16-14=2*, this will lead to *f(3,2)*.

Since $f(3, 2) = 0 \rightarrow x_3 = 0$.

So, optimal solution is: $(x_1, x_2, x_3) = (1,1,0)$, which yields the profit of 38.

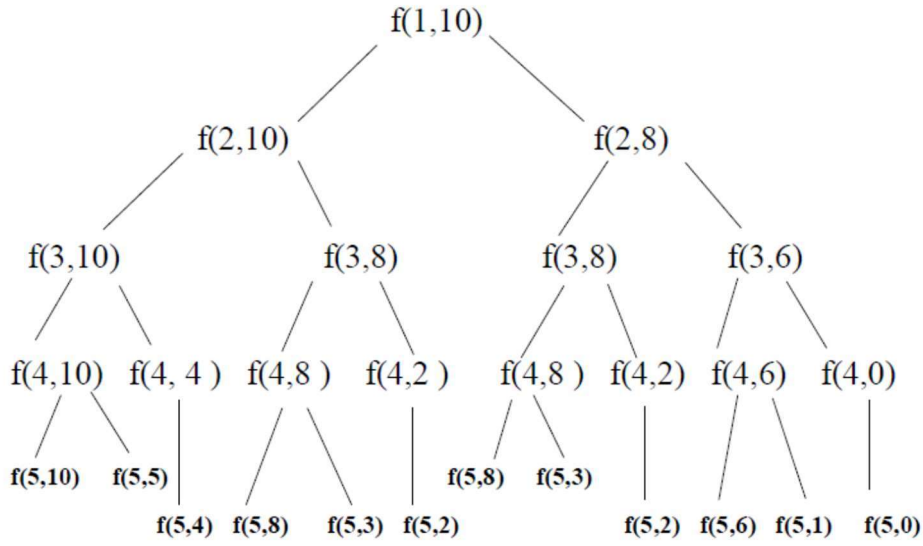
ALOGORITHM FOR 0/1 KNAPSACK PROBLEM USING FORWARD APPROCH:

```
Algorithm f(i, y)
{
    if(i=n) then
    {
        if(w[n]>y) then return 0;
        else return p[n];
    }
```

if(w[i]>y) **then return** f(i+1, y);
else return max [f(i + 1, y), (p_i + f(i + 1, $y - w_i$))];
}
→ INITIALLY THE ABOVE ALGORITHM IS INVOKED AS *f(1, M)*.

TIME COMPLEXITY:
→ let *T(n)* be the time this code takes to solve an instance with *n* objects.
So, $T(n) = \begin{cases} c, & if\ n = 1 \\ 2T(n-1) + c, & if\ n > 1 \end{cases}$
Solving this, we get time complexity equal to $O(2^n)$.
→ In general, if there are *d* choices for each of the *n* decisions to be made, there will be d^n possible decision sequences.

Example: Consider the case *n=5; M=10; (w_1, w_2, w_3, w_4, w_5) =(2,2,6,5,4); (p_1, p_2, p_3, p_4, p_5) = (6,3,5,4,6).*
Solution: To determine *f(1,10)*, function *f* is invoked as *f(1,10)*. The recursive calls made are shown by the following tree of recursive calls.



→ 27 invocations are done on *f*. We notice that several invocations redo the work of previous invocation. For example: *f(3,8), f(4,8), f(4,2), f(5,8), f(5,3), f(5,2)* are computed twice.
If we save the result of previous invocations, we can reduce the number of invocations to 21.

We maintain a table to store the values, as and when the function call computes there.
By using these table values, we can avoid re-computing the function, when it is again invoked.
When the recursive program is designed to avoid the re-computation, the complexity is drastically reduced from exponential to polynomial.

Recurrence equation for 0/1 knapsack problem using backward approach:

Let *f(i, y)* denote the value (or, profit) of an optimal solution to the knapsack instance with remaining capacity *y* and remaining objects *i, i-1, ... ,1* for which decisions have to be taken.
It follows that

$$f(1,y) = \begin{cases} p_1, & if\ w_1 \le y \\ 0, & if\ w_1 > y \end{cases} \quad \text{-------(1)}$$

and

$$f(i,y) = \begin{cases} max\left(f(i-1,y),\ p_i + f(i-1,y-w_i)\right), & if\ w_i \le y \\ f(i-1,y), & if\ w_i > y \end{cases} \quad \text{--(2)}$$

→ *f(n, M)* gives the value (or profit) of the optimal solution by including objects *n, n-1, ..., 1*.

Example: Let us determine *f(n, M)* recursively for the following 0/1 knapsack instance: *n=3; (w_1, w_2, w_3) = (2,3,4); (p_1, p_2, p_3) = (1,2,5) and M=6.*

Solution:

f(3,6) =max{f (2, 6), 5 + f(2, 2)}
f(2,6) =max{f (1, 6), 2 + f(1, 3)}
f(1,6) = p1 = 1
f(1,3) = p1 = 1
f(2,6) = max{1, 2+1} = 3
f(2,2) = f(1,2) = f(1,2) = 1
Now,
f(3,6) = max{3, 5+1} = 6
Tracing back values of x_i:
f(3,6) = 6
f(2,6) = 3
Since *f(3,6) ≠ f(2,6),* x_3*=1.*
After including object *3*, remaining capacity becomes *6-4= 2*, this willlead to the problem *f(2,2).*
f(2,2) = 1
f(1,2) = 1
Since *f(2,2) = f(1,2),* x_2*=0,* and this will lead to the problem *f(1,2).*

Since *f(1,2) = 1, x_1= 1.*
So, optimal solution is: $(x_1, x_2, x_3) = (1,0,1)$.

Solving 0/1 knapsack problem using Sets of Ordered Pairs:

→Let s^i represent the possible state, resulting from the 2^i decision sequences for $x_1, x_2, x_3 \dots \dots x_i$.

→A state refers to a tuple (p_j, w_j), w_j being the total weight of objects included in the knapsack and p_j being the corresponding profit.

NOTE: $s^0 = \{(0,0)\}$

→To obtain s^{i+1} from s^i, we note that the possibilities for x_{i+1} are *1* and *0*.

→When $x_{i+1} = 0$, the resulting states are same as for s^i.

When x_{i+1} =1, the resulting states are obtained by adding (p_{i+1}, w_{i+1}) to each state in s^i.We call the set of these additional states s_1^i.

→Now s^{i+1}can be computed by merging the states in s^i *and* s_1^i together, i.e., $s^{i+1} = s^i \cup s_1^i$. *The states in* s^{i+1} *set should be arranged in the increasing order of profits.*

NOTE:

1. If s^{i+1} contains two pairs (p_j, w_j) and (p_k, w_k) with the property that $p_j \leq p_k$ and $w_j \geq w_k$, we say that (p_k, w_k) dominates (p_j, w_j) and the dominated tuple (p_j, w_j) can be discarded from s^{i+1}. This rule is called purging rule.
2. By this rule all duplicate tuples will also be purged.
3. We can also purge all pairs (p_j, w_j) with $w_j > M$.

Finally profit for the optimal solution is given by *p* value of the last pair in s^n set.

Tracing back values of x_i's:

→Suppose that (p_k, w_k) is the last tuple in s^n, then a set of 0/1 values for the x_i's can be determined by carrying out a search through the s^i sets.

→ if $(p_k, w_k) \in s^{n-1}$, then we will set $x_n = 0$.

If $(p_k, w_k) \notin s^{n-1}$, then $(p_k - p_n, w_k - w_n) \in s^{n-1}$and we will set $x_n = 1$. This process can be done recursively to get remaining x_i values.

Example: Consider the knapsack instance:

n = 3, $(w_1, w_2, w_3) = (2, 3, 4)$ and $(p_1, p_2, p_3) = (1,2,5)$, and *M = 6*. Generate the sets s^i and find the optimal solution.

Solution:

$s^0 = \{(0,0)\}$;

By including object 1,
s_1^0 = {(1,2)};
By merging s^0 *and* s_1^0 , we get
s^1 = {(0,0), (1,2)};
By including object 2,
s_1^1={(2,3), (3, 5)};
By merging s^1 *and* s_1^1, we get
s^2 = {(0, 0), (1, 2), (2, 3), (3, 5)};
By including object 3,
s_1^2= {(5, 4), (6, 6), (7, 7), (8, 9)} = {(5, 4), (6, 6)};
By merging s^2 *and* s_1^2, we get
s^3= {(0,0), (1,2), (2, 3), (3, 5), (5, 4), (6, 6)};
By applying the purging rule, the tuple (3, 5) will get discarded.
s^3={(0,0), (1,2), (2, 3), (5, 4), (6, 6)};
Tracing out the values of x_i :
→The last tuple in s^3 is (6,6) $\notin s^2$. So, x_3=1.
→The last tuple (6,6) of s^3came from a tuple $(6 - p_3, 6 - w_3)$ = (6-5, 6-4) = (1, 2) belonging to s^2.
→The tuple (1, 2) of s^2, is also present in s^1. So, $x_2 = 0$.
→The tuple (1, 2) of s^1, is not present in s^0. So, $x_1 = 1$.
So, the optimal solution is: (x_1, x_2, x_3) = (1,0,1)

Example: Consider the knapsack instance:
n = 5, $(w_1, w_2, w_3, w_4, w_5)$ *= (2, 2, 6, 5, 4)* and $(p_1, p_2, p_3, p_4, p_5)$ *= (6,3,5,4,6)*, and *M = 10*. Generate the sets s^i and find the optimal solution.

Sol:- given $n = 5$, $(w_1, w_2, w_3, w_4, w_5) = (2, 2, 6, 5, 4)$,
$(P_1, P_2, P_3, P_4, P_5) = (6, 3, 5, 4, 6)$, and $M = 10$.

$S^0 = \{(0,0)\}$.

$S_1^0 = \{(6,2)\}$.

$\Rightarrow S^1 = S^0 \cup S_1^0 = \{(0,0), (6,2)\}$.

$S_1^1 = \{(0+3, 0+2), (6+3, 2+2)\} = \{(3,2), (9,4)\}$

$\Rightarrow S^2 = S^1 \cup S_1^1 = \{(0,0), (6,2), (3,2), (9,4)\}$

$= \{(0,0), (3,2), (6,2), (9,4)\}$

$S^2 = \{(0,0), (6,2), (9,4)\}$

$S_1^2 = \{(0+5, 0+6), (6+5, 2+6), (9+5, 4+6)\}$

$= \{(5,6), (11,8), (14,10)\}$

$\Rightarrow S^3 = S^2 \cup S_1^2 = \{(0,0), (6,2), (9,4), (5,6), (11,8), (14,10)\}$

$= \{(0,0), (5,6), (6,2), (9,4), (11,8), (14,10)\}$

$S^3 = \{(0,0), (6,2), (9,4), (11,8), (14,10)\}$

$S_1^3 = \{(6+4, 0+5), (6+4, 2+5), (9+4, 4+5), (11+4, 8+5), (14+4, 10+5)\}$

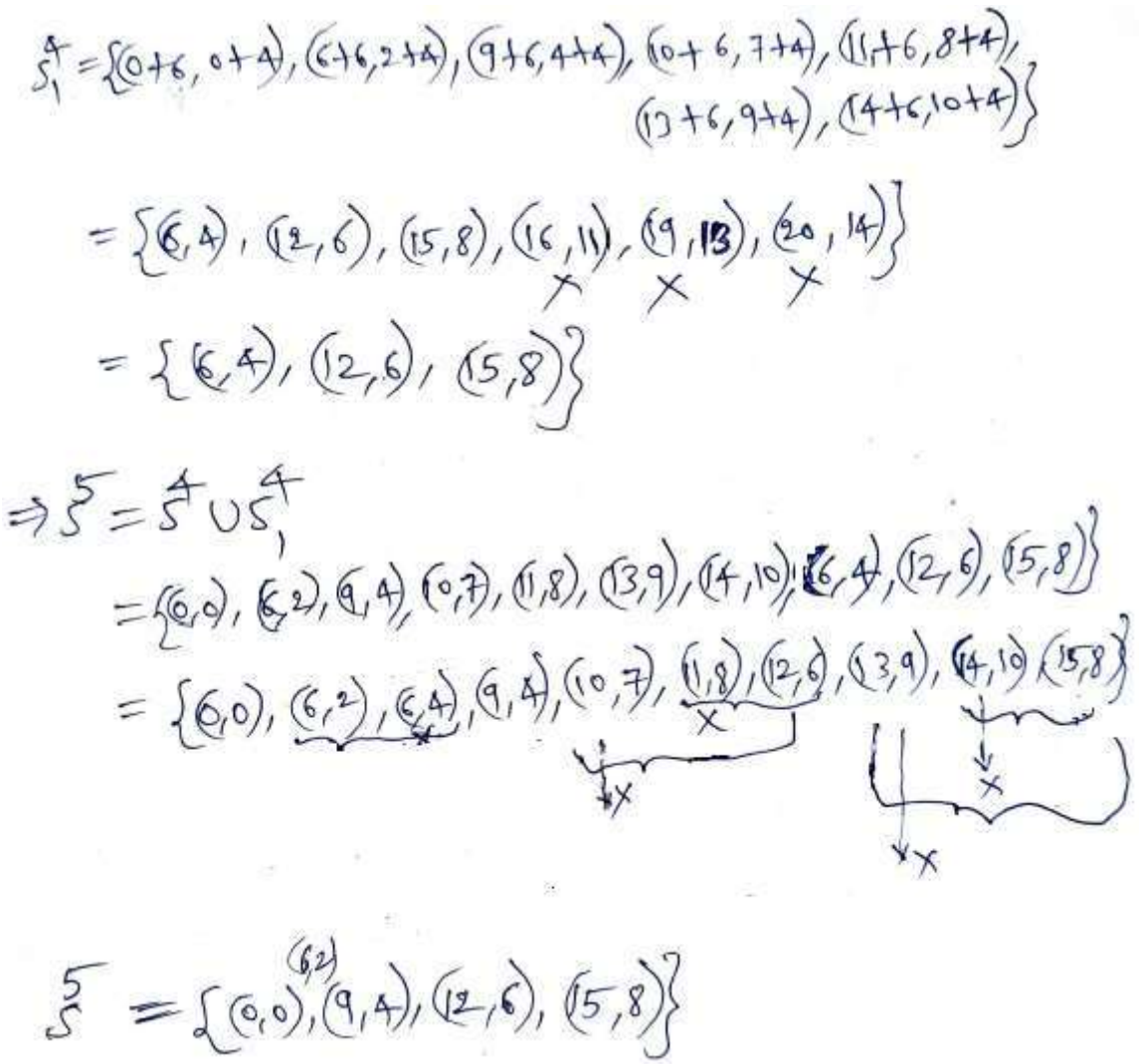
$= \{(4,5), (10,7), (13,9), (15,13), (18,15)\}$

$= \{(4,5), (6,7), (13,9),\}$

$\Rightarrow S^4 = S^3 \cup S_1^3 = \{(0,0), (6,2), (9,4), (11,8), (14,10), (4,5), (10,7), (13,9)\}$

$= \{(0,0), (4,5), (6,2), (9,4), (6,7), (11,8), (13,9), (14,10)\}$

$S^4 = \{(0,0), (6,2), (9,4), (6,7), (11,8), (13,9), (14,10)\}$



Tracing out the value of x_i:

→The last tuple in s^5 is (15,8) $\notin s^4$. So, x_5 = 1.

→The last tuple (15,8) of s^5came from a tuple $(15 - p_5, 8 - w_5)$ = (15-6, 8-4) = (9, 4) belonging to s^4.

→The tuple (9, 4) of s^4, is also present in s^3. So, $x_4 = 0$.

→The tuple (9, 4) of s^3, is also present in s^2. So, $x_3 = 0$.

→The tuple (9, 4) of s^2, is not present in s^1. So, $x_2 = 1$.

→The tuple (9, 4) of s^2 came from a tuple (9-p_2, 4-w_2) = (9-3, 4-2) = (6,2) belonging to s^1.

→ The tuple (6, 2) of s^1, is not present in s^0. So, $x_1 = 1$.

So, the optimal solution is: $(x_1, x_2, x_3, x_4, x_5)$ = (1,1,0,0,1).

How the dynamic-programming method works? (Not required for theory exam)
The dynamic-programming method works as follows.
Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly, we arrange for each subproblem to be solved only *once*, saving its solution. If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it. Dynamic programming thus uses additional memory to save computation time; it serves an example of a ***time-memory trade-off***. The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution.

A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach.

1. Top-down with memoization:

In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner and stores the result in the table.

2. Bottom-up method:

This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Solution to 0/1 Knapsack problem using Top-down Dynamic Programming approach with Memoization:

→Let us consider the backward recursive equation.

$$f(1, y) = \begin{cases} p_1, & if\ y \geq w_1 \\ 0, & if\ y < w_1 \end{cases}$$

$$f(i, y) = \begin{cases} max\ (f(i-1, y),\ (p_i + , f(i-1, y-w_i))), if\ y \geq w_i \\ f(i-1,\ y) \qquad , if\ \ y < w_i \end{cases}$$

→Let us rewrite the above backward recursive equation as follows:
$f(0, y) = 0$ for $y \geq 0$ and $f(i, 0) = 0$ for $i \geq 0$

$$f(i, y) = \begin{cases} max\ (f(i-1, y),\ (p_i + , f(i-1, y-w_i))), if\ y \geq w_i \\ f(i-1,\ y) \qquad , if\ \ y < w_i \end{cases}$$

→Initially this function is invoked as *f(n, M)*. This problem's size is *n* (i.e., the number of objects on which decisions have to be taken). Next, it calls the subproblem *f(n-1, y)* whose size is *n-1*, and so on. This recursion is repeated until a problem of smallest size i.e., *f(1, y)* is called.

Algorithm f(i, y)
{
// Let T[0:n, 0:M] be a global two dimensional array whose elements are
// initialized with -1 except for row 0 and column 0 which are initialized with 0's.
```
    if (T[i, y] <0) then      // if f(i, y) has not been computed previously
    {
        if(w[i]>y) then
        {
            T[i, y] := f(i-1, y);
        }
        else
        {
            T[i, y] := max(f(i-1, y), p[i] + f(i-1, y-w[i]));
        }
    }
    return T[i, y];
}
```
→Initially this function is invoked as *f(n, M)*.

What is the time and space complexity of the above solution?

Since our memoization array *T[0:n, 0:M]* stores the results for all the subproblems,

we can conclude that we will not have more than *(n+1)*(M+1)* subproblems (where '*n*' is the number of items and '*M*' is the knapsack capacity). This means that the time complexity will be *O(n*M)*.
The above algorithm will be using *O(n*M)* space for the memoization array *T*. Other than that, we will use *O(n)* space for the recursion call-stack. So, the total space complexity will be *O(n*M+n)*, which is asymptotically equivalent to *O(n*M)*.

Tracing back values of x[i]:

```
for i:=n to 1 step -1 do
{
  if T[i, y]=T[i-1, y] then
        x[i]:=0;
  else
  {
       x[i]:=1;
       y:=y-w[i];
   }
}
```

Example: consider the case n=5; M=10; $(w_1, w_2, w_3, w_4, w_5)$ = (2,2,6,5,4) ; $(p_1, p_2, p_3, p_4, p_5)$ = (6,3,5,4,6)
f(5,10) = max(f(4,10), 6+f(4,6))
f(4,10) = max(f(3,10), 4+f(3,5))
f(3,10) = max(f(2,10), 5+f(2,4))
f(2,10) = max(f(1,10), 3+f(1,8))

f(1,10) = max(f(0,10), 6+f(0,8)) = max(0, 6+0) = 6
f(1,8) = max(f(0,8), 6+f(0,6)) = max(0, 6+0) = 6
Now, f(2,10) = max(6, 3+6) = 9
f(2,4) = max(f(1,4), 3+f(1,2)
f(1,4) = max(f(0,4), 6+f(0,2)) = max(0, 6+0) = 6
f(1,2) = max(f(0,2), 6+f(0,0)) = max(0, 6+0) = 6
Now, f(2,4) = max(6,3+6) = 9
Now, f(3,10) = max(6, 5+6) = 11
f(3,5) = f(2,5)
f(2,5) = max(f(1,5), 3+f(1,3))
f(1,5) = max(f(0,5), 6+f(0,3)) = max(0, 6+0) = 6

f(1,3) = max(f(0,3), 6+f(0,1)) = max(0, 6+0) = 6
Now, f(2,5) = max(6, 3+6) = 9
So, f(3,5) = 9
Now, f(4,10) = max(11, 4+9) = 13
f(4,6) = max(f(3,6), 4+f(3,1))
f(3,6) = max(f(2,6), 5+f(2,0))
f(2,6) = max(f(1,6), 3+f(1,4))
f(1,6) = max(f(0,6), 6+f(0,4)) = max(0, 6+0) = 6
f(1,4) = 6 (Obtained from the table)
Now, f(2,6) = max(6, 3+6) = 9.
f(2,0) = 0
Now, f(3,6) = max(9, 5+0) = 9.
f(3,1) = f(2,1) = f(1,1) = 0
f(4,6) = max(9, 4+0) = 9
Now, f(5,10) = max(13, 6+9) =15.

Column indices (y values)

Row indices (i values)	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	-1(0)	-1(6)	-1(6)	-1(6)	-1(6)	-1(6)	-1	-1(6)	-1	-1(6)
2	0	-1(0)	-1	-1	-1(9)	-1(9)	-1(9)	-1	-1	-1	-1(9)
3	0	-1(0)	-1	-1	-1	-1(9)	-1(9)	-1	-1	-1	-1(14)
4	0	-1	-1	-1	-1	-1	-1(9)	-1	-1	-1	-1(14)
5	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1(15)

The optimal solution is: (x1, x2, x3, x4, x5) = (1, 1, 0, 0, 1).

Solution to 0/1 Knapsack problem using Bottom-up Dynamic Programming approach:

→Let us consider backward recursive equation as follows:
f(0, y) = 0 for y≥0 and f(i, 0) = 0 for i≥0

$$f(i, y) = \begin{cases} max\ (f(i-1, y),\ (p_i + ,f(i-1, y-w_i))), if\ y \geq w_i \\ f(i-1,\ y) \qquad\quad , if\ \ y < w_i \end{cases}$$

Step-01:

- Draw a table say '*T*' with *(n+1)* number of rows and *(M+1)* number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown below:

Column indices (y values)

Row indices (i values)

	0	1	2	3	…	M
0	0	0	0	0	…	0
1	0					
2	0					
3	0					
…	…					
N	0					

Step-02:
Start filling the table row wise top to bottom from left to right.
Use the following formula:

$T[i, y] = max\ \{ T[i-1, y], p_i + T[i-1, y-w_i] \}, if\ y \geq w_i$

$= T[i-1, y], if\ \ y < w_i$

Here, *T[i , y]* = maximum profit earned by taking decisions on items *1* to *i* with remaining capacity *y*.

- This step leads to completely filling the table.
- Then, value of the last cell (i.e., intersection of last row and last column) represents the maximum possible profit that can be earned.

Step-03:

To identify the items that must be put into the knapsack to obtain that maximum profit (that means to trace back the values of *xi*),

- Consider the last entry (i.e., cell) of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Algorithm:

```
Algorithm KnapSack(n, M)
{
// Let T[0:n, 0:M] be a global two dimensional array whose elements in row 0 and
// column 0 are initialized with 0's.
        for i:=1 to n do
        {
                for y:=1 to M do
                {

                        if(w[i]>y) then
                        {
                                T[i, y] := T[i-1, y];
                        }
                        else
                        {
                                T[i, y] := max(T[i-1, y], p[i] + T[i-1, y-w[i]]);
                        }
                }
        }

        return T[n, M];
}
```

→This function is invoked as KnapSack(n, M).

Time and Space Complexity:

Each entry of the table requires constant time $\theta(1)$ for its computation.

It takes $\theta(nM)$ time to fill $(n+1)(M+1)$ table entries. This means that the time complexity will be $O(n*M)$. Even though it appears to be polynomial time but actually it is called *pseudo polynomial time* because when $M \geq 2^n$, the time complexity is actually exponential but not polynomial.

The space complexity is $\theta(nM)$.

Tracing back values of x[i]:

```
for i:=n to 1 step -1 do
{
  if T[i, y]=T[i-1, y] then
        x[i]:=0;
  else
   {
       x[i]:=1;
       y:=y-w[i];
    }
}
```

Example: consider the case n=5; M=10; ; $(w_1, w_2, w_3, w_4, w_5)$ = (2,2,6,5,4) ; $(p_1, p_2, p_3, p_4, p_5)$ = (6,3,5,4,6)

Column indices (y values)

Row indices (i values)		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
w1=2, p1=6	1	0	0	6	6	6	6	6	6	6	6	6
w2=2, p2=3	2	0	0	6	6	9	9	9	9	9	9	9
w3=6, p3=5	3	0	0	6	6	9	9	9	9	11	11	14
w4=5, p4=4	4	0	0	6	6	6	9	9	10	11	13	14
W5=4, p5=6	5	0	0	6	6	6	9	12	12	12	15	15

The optimal solution is: (x1, x2, x3, x4, x5) = (1, 1, 0, 0, 1).

MATRIX CHAIN MULTIPLICATION:

→ Given a sequence of matrices we need to find the most efficient way to multiply those matrices together.

→ Since the matrix multiplication is associative, we can multiply a chain of matrices in several ways.

→ Here, we are actually not interested in performing the multiplication, but in determining in which order the multiplication has to be performed.

→ Let A be an $m \times n$ matrix and B be an $n \times p$ matrix.

→ The number of scalar multiplications needed to perform A*B is m*n*p. We will use this number as a measure of the time (or, cost) needed to multiply two matrices.

→ Suppose we have to compute the matrix product $M_1*M_2*\ldots*M_n$, where M_i has dimensions $r_i \times r_{i+1}$, $1 \le i \le n$.

$M_1 : r_1 \times r_2$

$M_2 : r_2 \times r_3$

.

.

.

$M_n : r_n \times r_{n+1}$

→ We have to determine the order of multiplication that minimizes the total number of scalar multiplications required.

→ Consider the case n=4. The matrix product $M_1*M_2*M_3*M_4$ may be computed in any of the following 5 ways.

1. $M_1*((M_2*M_3)*M_4)$
2. $M_1*(M_2*(M_3*M_4))$
3. $(M_1*M_2)*(M_3*M_4)$
4. $((M_1*M_2)*M_3)*M_4$
5. $(M_1*(M_2*M_3))*M_4$

Example:
Consider three matrices A_{2x3}, B_{3x4}, C_{4x5}.
The product A*B*C can be computed in two ways: (AB)C and A(BC).
The cost of performing (AB)C is: 2*3*4+2*4*5 = 64.
The cost of performing A(BC) is: 3*4*5+2*3*5 = 90.
So, the optimal (i.e., best) order of multiplication is (AB)C.

(OR)

The best way of parenthesizing the given matrix chain multiplication ABC is, (AB)C.

→The number of different ways in which the product of n matrices may be computed, increases exponentially with n, that is $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$. As a result, the brute force method of evaluating all the multiplication schemes and select the best one is not practical for large n.

DYNAMIC PROGRAMING FORMULATION:

→ We can use dynamic programming to determine an optimal sequence of pair wise matrix multiplications. The resulting algorithm runs in only $O(n^3)$ time.

→ Let $M_{i,j}$ denote the result of the product chain $M_i * M_{i+1} * \ldots * M_j$, i<j.

Ex: $M_1*M_2*M_3 = M_{1,3}$.

Thus $M_{i,i} = M_i$, $1 \leq i \leq n$.

Clearly $M_{i,j}$ has dimensions: $r_i \times r_{j+1}$.

→ Let c(i, j) be the cost of computing $M_{i,j}$ in an optimal way.Thus c(i, i) = 0, $1 \leq i \leq n$.

→Now in order to determine how to perform the multiplication $M_{i,j}$ optimally, we need to make decisions. What we want to do is to break the problem into sub problems of similar structure.

→ In parenthesizing the matrix multiplication, we can consider the highest level (i.e., last level) of parenthesization. At this level, we simply multiply two matrices together. That is, for any k, (i≤k<j),

$M_{i,j} = M_{i,k} * M_{k+1,j}$

→ Thus, the problem of determining the optimal sequence of multiplications is

broken up into two questions:
1. How do we decide where to split the chain (i.e., what is k)?
2. How do we parenthesize the sub chains $(M_i * M_{i+1} * \ldots * M_k)$ and $(M_{k+1} * \ldots * M_j)$?

→ In order to compute $M_{i,j}$ optimally, $M_{i,k}$ and $M_{k+1,j}$ should also becomputed optimally. Hence, the principle of optimality holds.

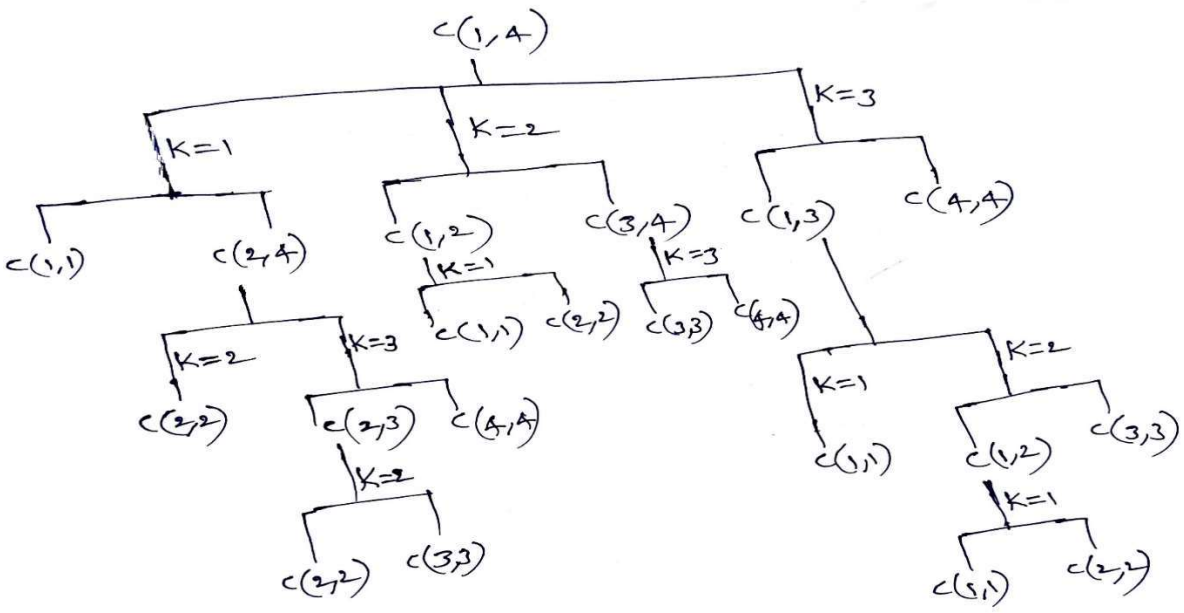
→ The cost of computing $M_{i,k}$ (i.e., $M_i * M_{i+1} * \ldots * M_k$) optimally is $c(i, k)$. The cost of computing $M_{k+1,j}$ (i.e., $M_{k+1} * \ldots * M_j$) optimally is $c(k+1, j)$.
→Since $M_{i,k}$ has dimensions $r_i \times r_{k+1}$ and $M_{k+1,j}$ has dimensions $r_{k+1} \times r_{j+1}$, the cost of multiplying the two matrices $M_{i,k}$ and $M_{k+1,j}$ is, $r_i * r_{k+1} * r_{j+1}$.
→ So, the cost of computing $M_{i,j}$ optimally is,
$c(i, j) = c(i, k) + c(k+1, j) + r_i * r_{k+1} * r_{j+1}$.
→ But, in the above equation, k (which is the splitting point of matrix product chain) can take different values based on the inequality condition $i \leq k < j$.

→ This suggests the following recurrence equation for computing $c(i, j)$:

$$c(i, j) = \begin{cases} 0, & if\ i = j \\ min_{i \leq k < j}\{c(i,k) + c(k+1,j) + r_i * r_{k+1} * r_{j+1}\}, & if\ i < j \end{cases}$$

→ The above recurrence equation for c may be solved recursively. The k value which results in $c(i, j)$ is denoted by $kay(i, j)$.
→ $c(1, n)$ is the cost of the optimal way to compute the matrix product chain $M_{1,n}$. And $kay(1, n)$ defines the last product to be done or where the splitting is done.
→ The remaining products can be determined by using kay values.

The tree of recursive calls of $c(\ )$ function for the matrix product chain $M_1 * M_2 * M_3 * M_4$:



SOLUTION FOR MATRIX CHAIN MULTIPLICATION:

→ The dynamic programing recurrence equation for *c* may be solved by computing each *c(i, j)* and *kay(i, j)* values exactly once in the order *j-i =1, 2, 3, .., n-1.*

Example: Apply dynamic programing technique for finding an optimal order of multiplying the five matrices with *r =(10, 5,1,10,2,10).*

Solution:

Initially, we have $c_{ii}=0$ and $kay_{ii}=0,\ 1\le i \le 5.$

Using the equation:

$$c(i,j)=\begin{cases}0, & if\ i=j\\ min_{i\le k<j}\{c(i,k)+c(k+1,j)+r_i * r_{k+1} * r_{j+1}\}, & if\ i<j\end{cases}$$

For $j-i=1$:

$c(1,2)=min_{1\le k<2}\{c(1,1)+c(2,2)+r_1 * r_2 * r_3\}$

$=0+0+10*5*1=50$

$kay(1,2)=1$

$c(2,3)=min_{2\le k<3}\{c(2,2)+c(3,3)+r_2 * r_3 * r_4\}$

$=0+0+5*1*10=50$

$kay(2,3)=1$

$c(3,4)=min_{3\le k<4}\{c(3,3)+c(4,4)+r_3 * r_4 * r_5\}$

$=0+0+1*10*2=20$
$kay(3,4)=3$

$c(4,5)=min_{4\leq k<5}\{c(4,4)+c(5,5)+r_4*r_5*r_6\}$
$=0+0+10*2*10=200$
$kay(4,5)=4$

For j – i = 2:

$c(1,3)=min_{1\leq k<3}\{c(1,1)+c(2,3)+r_1*r_2*r_4;\ c(1,2)+c(3,3)+r_1*r_3*r_4\}$
$=min\{0+50+10*5*10,\ 50+0+10*1*10\}$
$=min\{550,150\}$
$= 150$
$kay(1,3)=2$
$c(2,4)=min_{2\leq k<4}\{c(2,2)+c(3,4)+r_2*r_3*r_5;\ c(2,3)+c(4,4)+r_2*r_4*r_5\}$

$=min\{0+20+5*1*2,\ 50+0+5*10*2\}$
$=min\{30,\ 150\}$
$= 30\ kay(2,4)=2$

$c(3,5)=min_{3\leq k<5}\{c(3,3)+c(4,5)+r_3*r_4*r_6;\ c(3,4)+c(5,5)+r_3*r_5*r_6\}$
$=min\{0+200+1*10*10,\ 20+0+1*2*10\}$
$=min\{300,\ 40\}$
$= 40\ kay(3,5)=4$

For j – i = 3:

$c(1,4)=min_{1\leq k<4}\{c(1,1)+c(2,4)+r_1*r_2*r_5;\ c(1,2)+c(3,4)+r_1*r_3*r_5;\ c(1,3)+c(4,4)+r_1*r_4*r_5\}$
$=min\{0+30+10*5*2,\ 50+20+10*1*2,\ 150+0+10*2*2\}$
$=min\{130,\ 90,\ 190\}$
$= 90\ kay(1,4)=2$

$c(2,5)=min_{2\leq k<5}\{c(2,2)+c(3,5)+r_2*r_3*r_6;\ c(2,3)+c(4,5)+$

$r_2 * r_4 * r_6$; $c(2,4) + c(5,5) + r_2 * r_5 * r_6$}
=min{0+40+5*1*10, 50+200+5*10*10, 30+0+5*2*10}
=min{90,750,130}
= 90 kay(2,5)=2

$c(1,5)=min_{1 \le k < 5}$\{$c(1,1) + c(2,5) + r_1 * r_2 * r_6$; $c(1,2) + c(3,5) + r_1 * r_3 * r_6$; $c(1,3) + c(4,5) + r_1 * r_4 * r_6$; $c(1,4) + c(5,5) + r_1 * r_5 * r_6$\}
=min{0+90+10*5*10, 50+40+10*1*10, 150+200+10*10*10, 90+0+10*2*10}
=min{590, 190, 1350, 290}
= 190

kay(1,5)=2

j-i ⇩ \ i ⇨	1	2	3	4	5
0	$c_{11} = 0$ $kay_{11} = 0$	$c_{22} = 0$ $kay_{22} = 0$	$c_{33} = 0$ $kay_{33} = 0$	$c_{44} = 0$ $kay_{44} = 0$	$c_{55} = 0$ $kay_{55} = 0$
1	$c_{12} = 50$ $kay_{12} = 1$	$c_{23} = 50$ $kay_{23} = 2$	$c_{34} = 20$ $kay_{34} = 3$	$c_{45} = 200$ $kay_{45} = 4$	
2	$c_{13} = 150$ $kay_{13} = 2$	$c_{24} = 30$ $kay_{24} = 2$	$c_{35} = 40$ $kay_{35} = 4$		
3	$c_{14} = 90$ $kay_{14} = 2$	$c_{25} = 90$ $kay_{25} = 2$			
4	$c_{15} = 190$ $kay_{15} = 2$				

→ If k is the splitting point, $M_{i,j} = M_{i,k} * M_{k+1,j}$.
→ From the above table, the optimal multiplication sequence has cost 190. The sequence can be determined by examining *kay(1,5)*, which is equal to *2*.
→ So, $M_{1,5} = M_{1,2} * M_{3,5} = (M_1 * M_2) * (M_3 * M_4 * M_5)$
Since $kay(3,5) = 4$; $M_{3,5} = M_{3,4} * M_{5,5} = (M_3 * M_4) * M_5$
So, the optimal order of matrix multiplication is:
$M_{1,5} = M_1 * M_2 * M_3 * M_4 * M_5 = (M_1 * M_2) * ((M_3 * M_4) * M_5)$

Algorithm:

```
Algorithm MATRIX-CHAIN-ORDER(r)
{
    n := length(r) -1;  // n denotes number of matrices
    for i:= 1 to n do
        c[i, i]:= 0;
    for l := 2 to n   do      // l is the chain length
    {
        for i := 1 to n-l+1 do  // n-l+1 gives number of cells in the current row
        {
            j := i+l-1;
            c[i, j] := ∞;
            for k := i to j-1 do
            {
                q := c[i, k] + c[k+1, j] + r_i*r_{k+1}*r_{j+1};
                if q < c[i, j] then
                {
                    c[i, j]:= q;
                    kay[i, j]:= k;
                }
            }
        }
    }
    return c and kay;
}
```

Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices.

The table *kay[1:n, 2:n]* gives us the information we need to do so. Each entry *kay[i,j]* records a value of k such that an optimal parenthesization of $M_i * M_{i+1} * \ldots * M_j$ splits the product between M_k and M_{k+1}.

Thus, we know that the final matrix multiplication in computing $M_{1,n}$ optimally is $M_{1,k} * M_{k+1,n} = M_{1,kay(i,j)} * M_{kay(i,j)+1,n}$ We can determine the earlier matrix multiplications recursively, since *kay[1, kay[1,n]]* determines the last matrix multiplication when computing $M_{1,kay(i,j)}$ and *kay[kay[1,n]+1, n]* determines the last matrix multiplication when computing $M_{kay(i,j)+1,n}$. The following recursive

procedure prints an optimal parenthesization of $M_i*M_{i+1}*...*M_j$, given the *kay* table computed by MATRIX-CHAIN-ORDER and the indices *i* and *j*. The initial call PRINT-OPTIMAL-PARENS*(kay, i, n)* prints an optimal parenthesization of $M_1*M_{i+1}*...*M_n$.

```
Algorithm PRINT-OPTIMAL-PARENS(kay, i, j)
{
        if (i = j) then
                print "M"i;
        else
        {
                print "(";
                PRINT-OPTIMAL-PARENS(kay, i, kay[i, j]);
                PRINT-OPTIMAL-PARENS(kay, kay[i, j]+1, j ];
                print ")"
        }
}
```

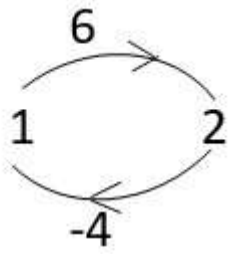
All-Pairs Shortest Paths problem: -

Given a graph *G*, we have to find a shortest path between every pair of vertices. That is, for every pair of vertices *(i, j)*, we have to find a shortest from *i* to *j*.

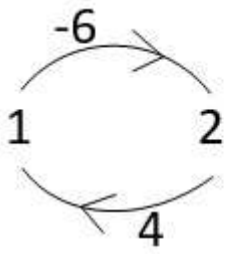
Let *G = (V, E)* be a weighted graph with *n* vertices. Let *c* be the cost adjacency matrix for G such that $c[i, i]=0, 1 \le i \le n$ and $c[i, j]=\infty$ if $i \neq j$ and $<i, j> \notin E(G)$.

When no edge has a negative length, the All-Pairs Shortest Paths problem may be solved by applying Dijkstra's greedy Single-Source Shortest Paths algorithm *n* times, once with each of the *n* vertices as the source vertex. This process results in $O(n^3)$ solution to the All-Pairs problem.

We will develop a dynamic programming solution called Floyd's algorithm which also runs in $O(n^3)$ time but works even when the graph has negative length edges (provided there are no negative length cycles).



Graph with positive cycle
Cycle length = 2
Floyd's algorithm works



Graph with negative cycle
Cycle length = -2
Floyd's algorithm doesn't work

We need to determine a matrix $A[1:n,1:n]$ such that $A[i, j]$ is the length of the shortest path from vertex i to vertex j.

The matrix A is initialized as $A^0[i, j] = c[i, j],\ 1 \le i \le n,\ 1 \le j \le n$. The algorithm makes n passes over A. In each pass, A is transformed. Let $A^1, A^2, A^3, \ldots, A^n$ be the transformations of A on the n passes.

Let $A^k[i, j]$ denote the length of the shortest path from i to j that has no intermediate vertex larger than k. That means, the path possibly passes through the vertices $\{1, 2, 3, \ldots, k\}$, but not through the vertices $\{k+1, k+2, \ldots, n\}$.

So, $A^n[i, j]$ gives the length of the shortest path from i to j because all intermediate vertices $\{1, 2, 3, \ldots, n\}$ are considered.

How to determine $A^k[i, j]$ for any $k \ge 1$?

A shortest path from i to j going through no vertex higher than k may or mayn't go through k.

If the path goes through k, then $A^k[i, j]=A^{k-1}[i, k]+A^{k-1}[k, j]$, following the principle of optimality.

If the path doesn't go through k, then no intermediate vertex has index greater than $(k-1)$.

Hence $A^k[i, j]=A^{k-1}[i, j]$.

By combining both cases, we get

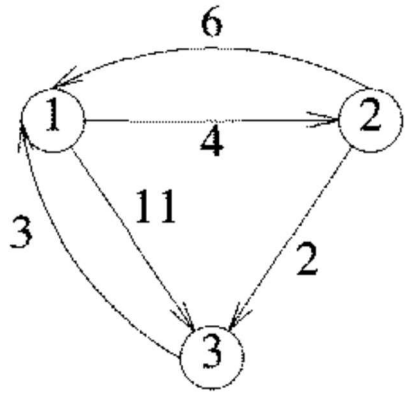
$A^k[i, j]=min\{ A^{k-1}[i, j], A^{k-1}[i, k]+A^{k-1}[k, j]\}, k\geq 1.$

This recurrence can be solved for A^n by first computing A^1, then A^2, then A^3, and so on. In the algorithm, the same matrix A is transformed over n passes and so the superscript on A is not needed. The k^{th} pass explores whether the vertex k lies on an optimal path from i to j, for all i and j. We assume that the shortest path *(i, j)* contains no cycles.

```
Algorithm Allpaths(c, A, n)
{
        //c[1:n,1:n] is the cost adjacency matrix of a graph with n vertices.
        //A[i, j] is length of a shortest path from vertex i to vertex j.
        //c[i, i]=0,  for 1≤i≤n.
        for i:=1 to n do
        for j:=1 to n do
                        A[i, j]=c[i, j];  //copy c into A
        for k:=1 to n do
        for i=1 to n do
                        for j=1 to n do
                                A[i, j]:=min(A[i, j], A[i, k]+A[k, j]);
}
```

Time complexity, $T(n)=O(n^3)$.

EXAMPLE: - Find out shortest paths between all pairs of vertices in the following digraph.



Step-1: $A^0 \leftarrow c$

0	4	11
6	0	2
3	∞	0

Step-2: Using, $A^1[i, j] = \min\{A^0[i, j], A^0[i, 1]+A^0[1, j]\}$

A^1

0	4	11
6	0	2
3	7	0

Step-3: Using, $A^2[i, j] = \min\{A^1[i, j], A^1[i, 2]+A^1[2, j]\}$

A^2

0	4	6
6	0	2
3	7	0

Step-4: Using, $A^3[i, j] = \min\{A^2[i, j], A^2[i, 3]+A^2[3, j]\}$

A^3

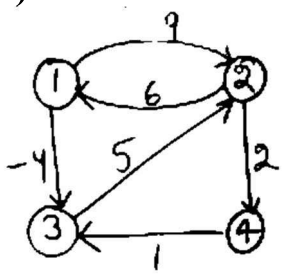
0	4	6
5	0	2
3	7	0

All-Pairs Shortest Paths:

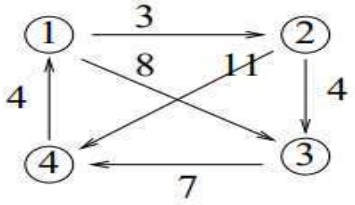
$1 \rightarrow 2$	$2 \rightarrow 3$	$3 \rightarrow 1$
$1 \rightarrow 2 \rightarrow 3$	$2 \rightarrow 3 \rightarrow 1$	$3 \rightarrow 1 \rightarrow 2$

Exercises:

(1)



(2)



The Traveling Salesperson problem: -

→Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

→Let the cities be represented by vertices in a graph and the distances between them be represented by weights on edges in the graph.

→Let $G(V, E)$ be the directed graph with n vertices.
→Let graph G be represented by cost adjacency matrix $C[1:n, 1:n]$.
→For simplicity, $C[i, j]$ is denoted by C_{ij}. If $<i, j> \notin E(G)$, $C_{ij}=\infty$; otherwise $C_{ij} \geq 0$.

→A tour of G is a directed cycle that includes every vertex in V. The cost of a tour is the sum of the costs of the edges on the tour.
→The travelling salesperson problem aims at finding an optimal tour (i.e., a tour of minimum cost).

→Let the given set of vertices be $\{1, 2, ..., n\}$. In our discussion, we shall consider a tour be a simple path that starts at vertex 1 and terminates at 1.

→Every possible tour can be viewed as consisting of an edge $<1, k>$ for some $k \in V-\{1\}$ and a path from vertex k to vertex 1.
The path from vertex k to vertex 1 goes through each vertex in the set $V-\{1, k\}$ exactly once.

→Suppose the tour consisting of the edge $<1, k>$ (for some $k \in V-\{1\}$) followed by a path from k to 1 is an optimal tour. Then the path from k to 1 should also be optimal. Thus, the principle of optimality holds.

→Let $g(i, S)$ denote the length of the shortest path starting at vertex i, going through all vertices in set S, and terminating at vertex 1. Thus $g(1, V-\{1\})$ gives the length of an optimal tour.
→If the optimal tour consists of the edge $<1, k>$, then from the principle

of optimality, it follows that $g(1, V-\{1\}) = C_{1k}+g(k, V-\{1,k\})$.
→But we don't know for which value of k the tour will be optimal. We know that k take can any value from the set $\{2, 3, \ldots, n\}$.
→The RHS of the above recurrence can be evaluated for each value of k and the minimum of those results should be assigned to $g(1, V-\{1\})$ resulting in the following recurrence equation:

$$g(1, V-\{1\}) = \min_{2\le k\le n} \{C_{1k}+g(k, V-\{1,k\})\} \text{ ------- (1)}$$

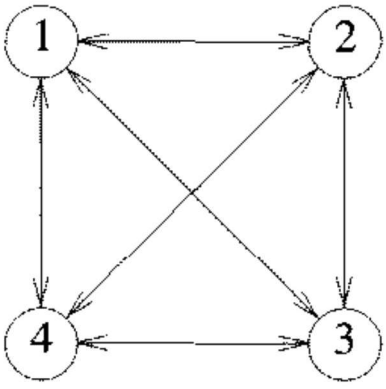
→Generalizing equation *(1)* for i not in S, we obtain

$$g(i, S)=\min_{\forall j\in S} \{C_{ij}+g(j, S-\{j\})\} \text{ ------ (2)}$$

→Equation *(1)* can be solved for $g(1, V-\{1\})$ if we know $g(k, V-\{1,k\})$ for all choices of k which can be obtained by using equation *(2)*.
→Clearly, $g(i, \emptyset)=C_{i1}, 1\le i\le n$.

→The optimal tour can be found by noting the vertex which resulted in minimum cost at each stage.
An algorithm that proceeds to find an optimal tour by making use of *(1)* and *(2)* will require
$O(n^2 2^n)$ time.

<u>EXAMPLE</u>: - Consider the directed graph below and its edge lengths given by cost adjacency matrix.



$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Find optimal sales person tour from vertex1.

Answer:-

$g(1,\{2,3,4\})=\min\{C_{12}+g(2,\{3,4\}),C_{13}+g(3,\{2,4\}),C_{14}+g(4,\{2,3\})\}$

$g(2,\{3,4\})=\min\{C_{23}+g(3,\{4\}),C_{24}+g(4,\{3\})$

$g(3,\{4\})=C_{34}+g(4,\emptyset)=C_{34}+C_{41}$

$=12+8=20.$

$g(4,\{3\})=C_{43}+g(3,\emptyset)=C_{43}+C_{31}$

$=9+6=15.$

$g(2,\{3,4\})=\min\{9+20,10+15\}$

$=25.$

$g(3,\{2,4\})=\min\{C_{32}+g(2,\{4\}),C_{34}+g(4,\{2\})\}$

$g(2,\{4\})=C_{24}+g(4,\emptyset)=C_{24}+C_{41}$

$=10+8=18.$

$g(4,\{2\})=C_{42}+g(2,\emptyset)=C_{42}+C_{21}$

$=8+5=13.$

$g(3,\{2,4\})=\min\{18+13,12+13\}$

$=\min\{31, 25\}$

$=25.$

$g(4,\{2,3\})=\min\{C_{42}+g(2,\{3\}),C_{43}+g(3,\{2\})\}$

$g(2,\{3\})=C_{23}+g(3,\emptyset)=C_{23}+C_{31}$

$=9+6=15.$

$g(3,\{2\})=C_{32}+g(2,\emptyset)=C_{32}+C_{21}$

$=13+5=28.$

$g(4,\{2,3\})=\min\{8+15,9+28\}$

$=\min\{23,37\}$

$=23.$

$g(1,\{2,3,4\})=\min\{10+25,15+25,20+23\}$

$=\min\{35,40,43\}$

$=35.$

$g(1,\{2,3,4\})=35.$

Construction of the optimal tour step by step:-

$g(1,\{2,3,4\})=C_{12}+g(2,\{3,4\})$ ➔ $1\rightarrow2$

$g(2,\{3,4\})=C_{24}+g(4,\{3\})$ ➔ $1\rightarrow2\rightarrow4$

$g(4,\{3\})=C_{43}+g(\{3,\emptyset\})$ ➔ $1\rightarrow2\rightarrow4\rightarrow3\rightarrow1$

So, the optimal tour is:- $1\rightarrow2\rightarrow4\rightarrow3\rightarrow1$.

UNIT-IV
BACKTRACKING

→ Backtracking: *Returning to a previous point.*

GENERAL METHOD: -

→ It is a technique used to solve many difficult combinatorial problems with a large search space, by systematically trying and eliminating different possibilities.

→ Any problem which deals with searching for a set of solutions or which asks for an optimal solution satisfying some constraints can be solved using the backtracking formulation.

→ In many applications of the backtrack method, the desired solution is expressed as an n-tuple or a vector (x_1, x_2, ..., x_n), where the values for x_i are chosen from some finite set S_i.

→ The solution vector is constructed by considering one component after another.

→ After considering the first choice of the next component, the partial solution vector is evaluated against a given criterion.

(i) If the criterion is satisfied, then the next component is included into the solution vector.

(ii) If the criterion is not satisfied, we will not include that component and there is no need of considering the remaining components also. In such a case, we backtrack by considering the next choice for the previous component of the partial solution vector.

→ In this way, the size of the solution search space of a problem can drastically reduce by using backtracking method when compared to exhaustive search.

State-Space Trees:

→ The process of obtaining a solution to a problem using backtracking can be illustrated with the help of a state space tree.

→ The nodes of the state space tree are generated in the depth first order beginning at the root node.

→ The nodes reflect specific choices made for components of a solution vector. The root node represents an initial state before the search for a solution begins.

→ The root node is at first level. The nodes at second level represent the choices made for the first component of a solution vector. The nodes at third level represent the choices made for the second component of a solution vector-, and so on.

→ The root node is considered both a live node and an E-node (i.e., expansion node). At any point of time, only one node is designated as an E-node. From the E-node, we try to move to (or, generate) a new node (i.e., child of E-node).

→ If it is possible to move to a child node from the current E-node (i.e., if there is any component yet to be included in the solution vector) then that child node will be generated by adding the first legitimate choice for the next component of a solution vector and the new node becomes a live node and also the new E-node. The old E-node remains as a live node.
→ At the newly generated node, we apply the constraint function (or, criterion) to determine whether this node can possibly lead to a solution. If it cannot lead to a solution (i.e., if it doesn't satisfy constraints) then there is no point in moving into any of its subtrees and so this node is immediately killed and we move back (i.e., backtrack) to the most recently seen live node (i.e., its parent) to consider the next possible choice for the previous component of a solution vector. If there is no such choice, we backtrack one more level up the tree, and so on. The final live node becomes new E-node.
→ Finally, if the algorithm reaches a complete solution, it either stops (if just one solution is required) or continues searching for other possible solutions.

N-QUEENS PROBLEM: -

→ We are given an *nxn* chessboard and we need to place *n* queens on the chess board such that they are non-attacking each another (i.e., no two queens should lie on the same row, or same column, or same diagonal).
→ Due to the first two restrictions, it is clear that each row and column of the board will have exactly one queen.
→ Let us number the rows and columns of the chessboard *1* through *n*. The queens can also be numbered *1* through *n*.
→ Since, each queen must be on a different row, we can assume that queen *i* will be placed on row *i*.
→ Each solution to the *n-queens* problem can therefore be represented as an n- tuple $(x_1, x_2, \ldots, x_n)$ where *xi* is the column number on which queen *i* is placed.
→ The x_i values should be distinct since no two queens can be placed on the same column.

HOW TO TEST WHETHER TWO QUEENS ARE ON SAME DIAGONAL?

→ We observe that all the elements on the same diagonal that runs from the upper-left to the lower-right have the same *row-column* value.
→ Also, all the elements on the same diagonal that runs from upper-right to lower-left have the same *row+column* value.
→ Suppose two queens are placed at positions *(i, j)* and *(k, l)*. Then they are on the same diagonal only if

i-j = k-l **(or)** *i+j = k+l*

By rearranging the terms, we have

j-l = i-k **(or)** *j-l = k-i*

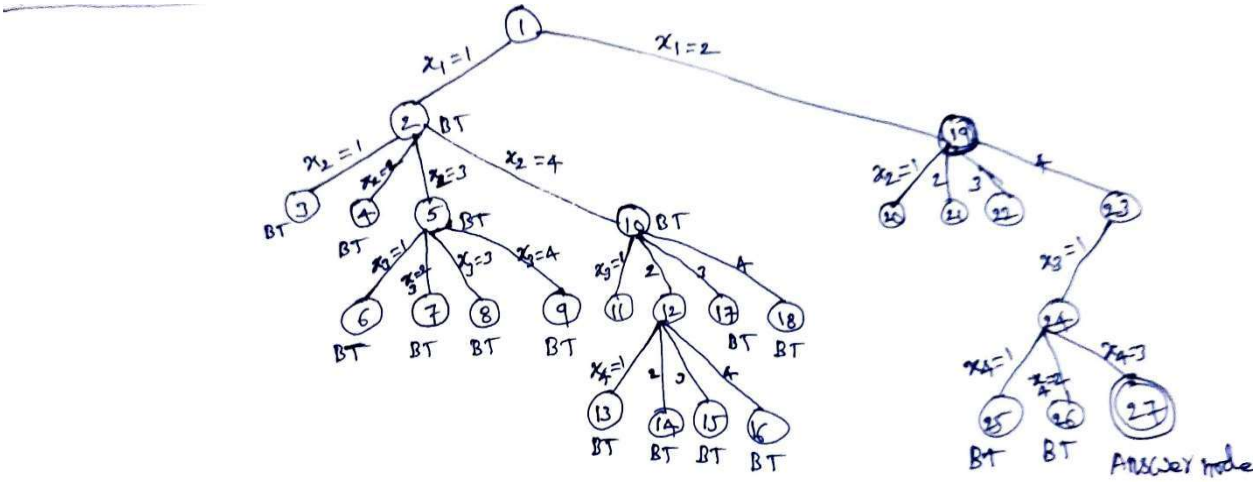
→ Therefore, two queens lie on the same diagonal if and only if |*j-l*| = |*i-k*|.

Algorithm for n-queens problem using backtracking:

```
Algorithm NQueens (k, n)
// using backtracking, this procedure prints all possible placements of n queens on
//an n×n chessboard so that they are non-attacking.
{
   for i := to n do   // for each column i from 1 to n
   {
       if (place(k, i)) then // check whether k^th queen can be placed on column i
       {
              x[k] :=i;
              if(k=n) then write(x[1: n]);
              else NQueens(k+1, n);
       }
   }
}
Algorithm place(k, i)
//returns true if a queen can be placed in k^th row and i^th column, otherwise it
//returns false.
// x[] is a global array whose first k-1 values have been set.
//Abs(r) returns absolute value of r.
{
       for j:=1 to k-1 do
       {
              if(x[j]=i) or (Abs(x[j]-i)=Abs(j-k)) then
                  return false;
       }
       return true;
}
```

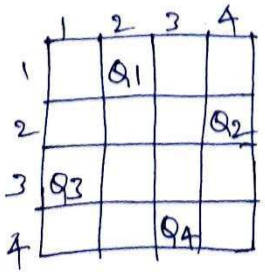
→ The **NQueens** algorithm is invoked as NQueens(1, n).

Portion of the State-space tree generated for solving 4-Queens problem: -



one of the solutions is:

$(x_1, x_2, x_3, x_4) = (2, 4, 1, 3)$



Graph Coloring (or) m-coloring problem: -

Given an undirected graph *G* and a positive integer *m*, determine if the graph can be colored with at most *m* colors such that no two adjacent vertices of the graph have the same color. This is called *m-coloring problem*. This is also known as *m-colorability decision problem.*

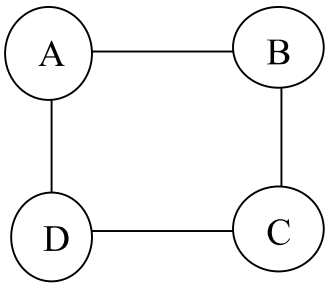
→ Here coloring of a graph means the assignment of colors to all vertices.

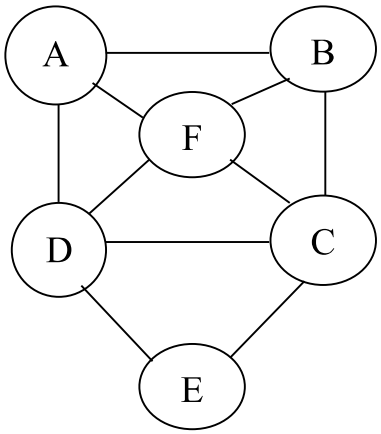
→ If the solution exists, then display which color is assigned to which vertex.

Side points:

1. *m-colorability optimization problem:* -
This problem asks for the smallest integer *m* for which the graph G can be colored. This integer is referred to as the chromatic number of the graph.
2. *Chromatic number of a graph:* - Minimum number of colors required to color a given graph such that no two adjacent vertices have same color.
3. If *d* is the maximum degree in a given graph, then its *chromatic number* ≤ *(d+1).*

Example: Find the chromatic number of the following graphs.

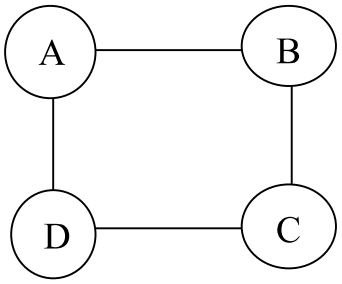




→ Let us number the vertices of the graph *1* through *n* and the colors *1* through *m.*

→ So, a solution to the graph coloring problem can be represented as an *n*-tuple *(x1, x2, ..., xn)* where *xi* is the color of vertex *i*.

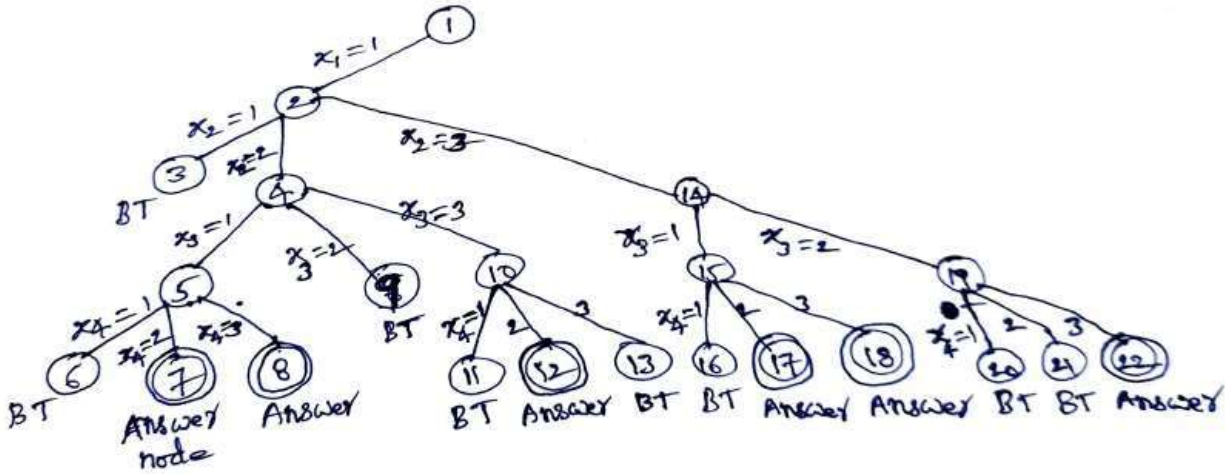
Problem: Find all possible ways of coloring the below graph with m=3.



Solution:

Let A=1, B=2, C=3, and D=4.

Portion of State-space tree generated for solving 3-coloring problem: -



Some of the solutions are: (x1, x2, x3, x4) = (1, 2, 1, 2)
= (1, 2, 1, 3)
= (1, 2, 3, 2)
= (1, 3, 1, 2)
= (1, 3, 1, 3)
= (1, 3, 2, 3)

Algorithm for m-coloring problem using backtracking:

→ Suppose we represent a graph by its adjacency matrix G[1:n, 1:n] where G[i, j]=1 if (i, j) is an edge and G[i, j]=0 otherwise.

→ Initially, the array x[] is set to zero.

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking scheme.
// The graph is represented by its Boolean adjacency matrix G[1:n,1:n].
// All possible assignments of 1, 2, ..., m to the vertices of the graph are printed,
// such that adjacent vertices are assigned distinct integers.
// k is the index of the next vertex to color.
{
        while(TRUE)
        {
            // Generate all legal assignments for x[k].
            NextValue(k);   // Assign a legal color to x[k].
            if (x[k] = 0) then return;   //No new color is possible
            if (k=n) then   //At most m colors have been used to color the n vertices.
                write (x[1:n]);
            else mColoring(k+1);
        }
}
Algorithm NextValue(k)
// x[1], ..., x[k - 1] have been assigned integer values in the range [1, m] such that
// adjacent vertices have distinct integers.
// A value for x[k] is determined in the range [0, m].
// x[k] is assigned the next highest numbered color while maintaining distinctness
// from the adjacent vertices of vertex k. If no such color exists, then x[k] is 0.
{
        while(TRUE)
        {
            x[k] := (x[k]+1) mod (m+1);   //Next highest color.
            if {x[k] = 0) then return;   // All colors have been used.
            for j:=1 to n do
            {
               // Check if this color is distinct from adjacent colors.
               if ((G[k,j]≠0) and (x[k] = x[j]))  //If (k, j) is an edge and if adjacent
                                                  // vertices have the same color.
               then break;
            }
            if (j=n+1) then return;   // New color found
        }   // Otherwise try to find another color.
}
```

→ **mColoring** algorithm is initially invoked as mColoring(1).

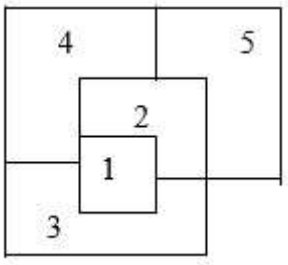
Planar graph:

→ A graph is said to be planar iff it can be drawn on a plane in such a way that no two edges cross each other.

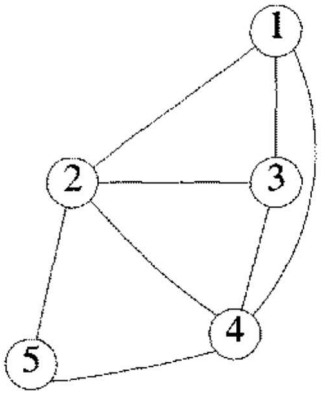
→ ***The chromatic number of a planar graph is not greater than 4.***

→ Suppose we are given a map, then it can be converted into planar graph as follows: Consider each region of the map as a node. If two regions are adjacent, then the corresponding nodes are joined by an edge.

→ Consider the following map with five regions:



The corresponding planar graph is:



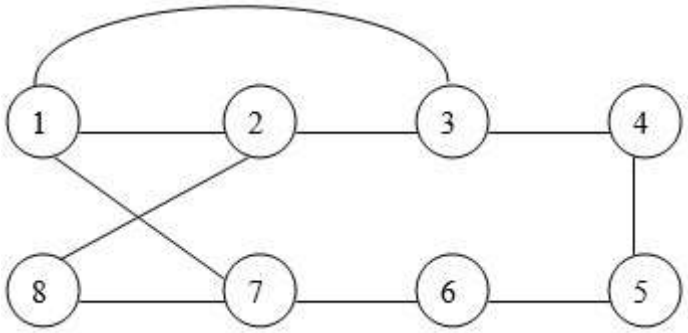
HAMILTONIAN CYCLES: -

→ Let $G = (V, E)$ be a connected graph with n vertices.

→ A Hamiltonian cycle is a round-trip path along n edges of G that visits every vertex once and returns to its starting position.

→ In other words, if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and v_i are distinct except for v_1 and v_{n+1} which are equal.

Example:
Consider the following graph:

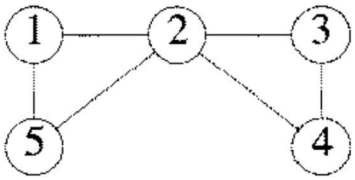


This graph has the following Hamiltonian cycle:

1—3—4—5—6—7—8—2—1

Note: If a graph has an articulation point, then there will be no Hamiltonian cycles.

Example: The following graph doesn't contain Hamiltonian cycle.



→ We can write a backtracking algorithm that finds all the Hamiltonian cycles in a graph. We will output only distinct cycles.

→ We assume that the vertices of the graph are numbered from *1* to *n*.

→ The backtracking solution vector $(x_1, x_2, ..., x_n)$ is defined so that x_i represents the i^{th} visited vertex of the proposed cycle.
→ The graph is represented by its adjacency matrix *G[1:n, 1:n]*.
→ *x[2:n]* are initialized to zero. And, *x[1]* is initialized to 1 because we assume that cycles start from vertex 1.
→ For $2 \le k \le n-1$, x_k can be assigned any vertex *v* in the range from *1* to *n* provided it is distinct from $x_1, x_2, ..., x_{k-1}$ and there exists an edge between *v* and x_{k-1}.
→ Now, x_n can be assigned the remaining vertex, provided there exists an edge to it from both x_1 and x_{n-1}.

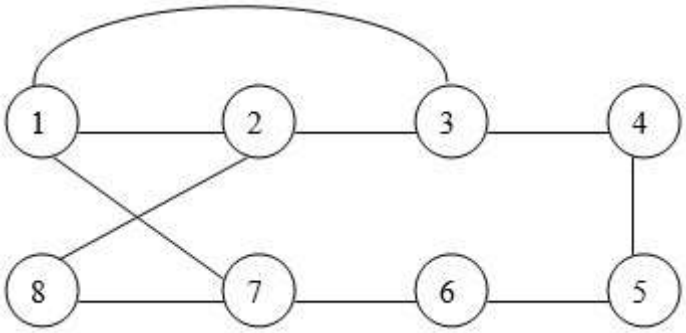
Algorithm for Hamiltonian Cycles problem using backtracking:

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of backtracking to find all the
// Hamiltonian cycles of a graph.
// The graph is stored as an adjacency matrix G[1:n, 1:n].
// All cycles begin at node 1.
{
        while(TRUE)
        {
                // Generate values for x[k].
                NextValue(k);       // Assign a legal next value to x[k].
                if (x[k] = 0) then return;
                if (k = n) then write (x[1:n]);
                else Hamiltonian(k+1);
        }
}


Algorithm NextValue(k)
// x[1: k - 1] is a path of k – 1 distinct vertices. If x[k] =0, then no vertex has as yet
// been assigned to x[k].
// After execution, x[k] is assigned to the next highest numbered vertex which
// does not already appear in x[1:k - 1] and is connected by an edge to x[k - 1].
// Otherwise, x[k] =0.
// If k = n, then in addition, x[k] is connected to x[1].
{
        while(TRUE)
        {
```

```
            x[k] := (x[k]+1) mod (n+1);      // Next vertex.
            if (x[k] = 0) then return;
            if (G[x[k-1], x[k]] ≠ 0) then // Is there an edge?
            {
                for j:= 1 to k – 1 do        // Check for distinctness.
                    if (x[j] = x[k]) then break;
                if (j = k) then        //If true, then the vertex is distinct.
                        if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))  then
                            return;
            }
        }
}
```

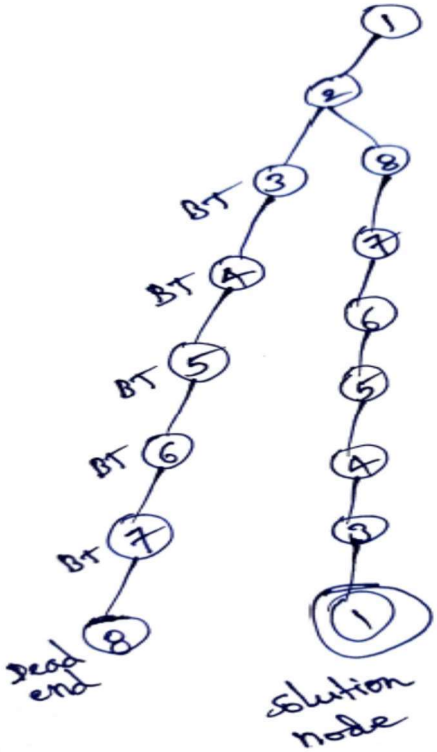
→ The algorithm **Hamiltonian** is initially invoked as Hamiltonian(2).

Example: **Find the Hamiltonian cycles for the following graph using backtracking.**

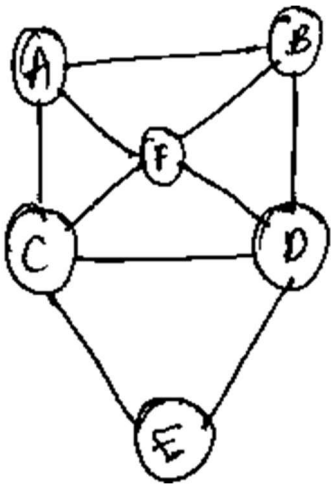


Solution:

Portion of State-space tree generated for solving Hamiltonian cycles problem: -

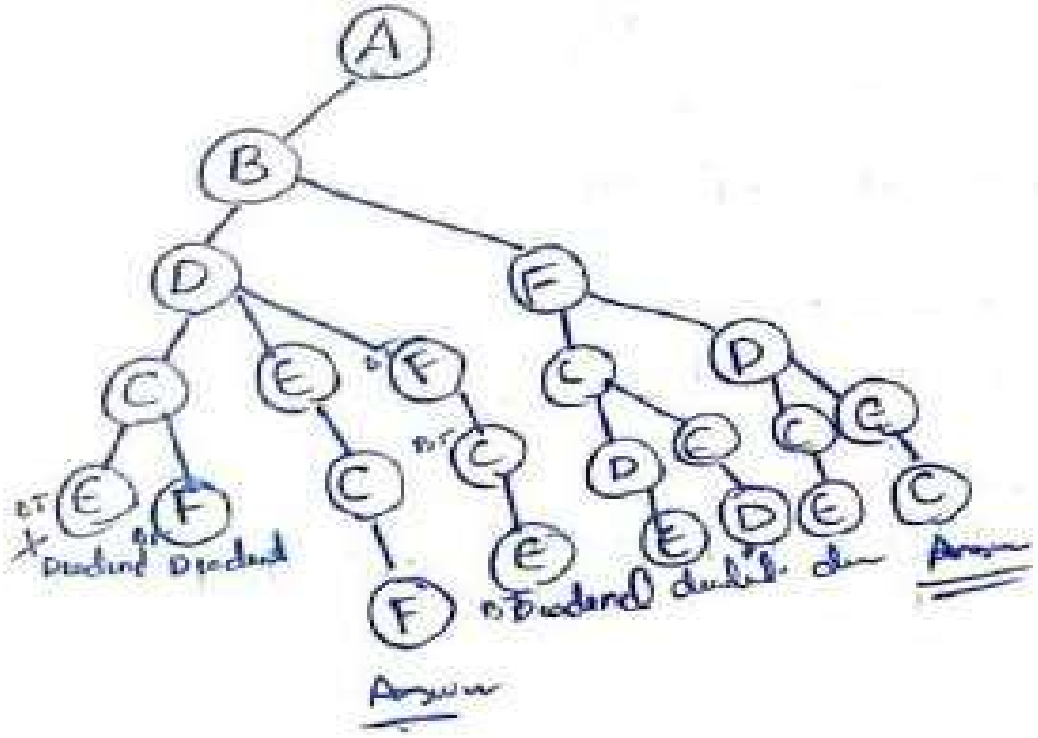


Example: **Find the Hamiltonian cycles for the following graph using backtracking.**



Solution:

Portion of State-space tree generated for solving Hamiltonian cycles problem: -



SUM OF SUBSETS: -

→ Suppose we are given n distinct positive numbers (usually called weights) w_i, $1 \le i \le n$ and we need to find all combination (subsets) of these numbers whose sums are equal to a given integer m.

→ Each solution subset is represented by an n-tuple $(x_1, x_2, \ldots, x_n)$ such that $x_i \in \{0,1\}$, $1 \le i \le n$.

If w_i is not included in subset, then $x_i = 0$.

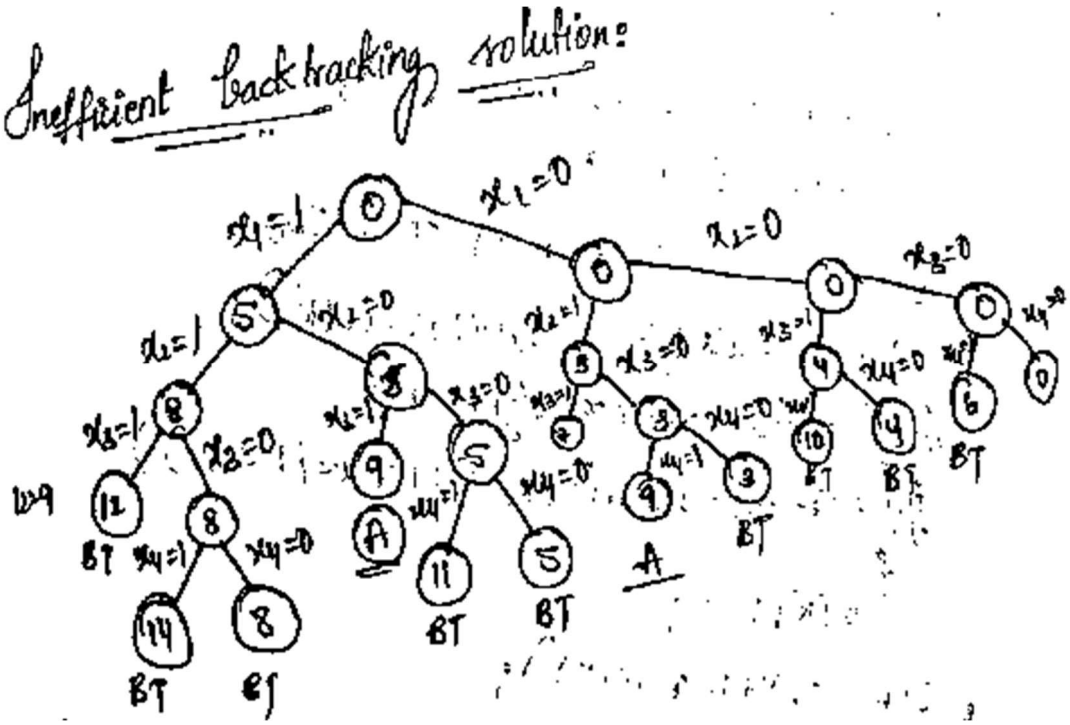
If w_i is included in subset, then $x_i = 1$.

Example: For $n=4$, $(w_1, w_2, w_3, w_4) = (5,3,4,6)$, and $m=9$,

$(x_1, x_2, x_3, x_4) = (1,0,1,0)$

$= (0,1,0,1)$

Size of solution search space is $2 \times 2 \times 2 \times 2 \times \ldots$ n times $= 2^n$.



Note: In the above figure, the number in the circle denotes the sum of the weights considered till now.

EFFICIENT BACKTRACKING SOLUTION FOR SUM OF SUBSETS: -

→ We assume that w_i's are initially in non-decreasing order.

→ At each stage, we have two choices for x_i's, i.e., *0* and *1*.

→ In the state space tree, for a node at level *i*, left child corresponds to $x_i=1$, right child corresponds to $x_i=0$.

→ Suppose we have fixed the values of $x_1, x_2, \ldots, x_{k-1}$. Now, if we choose $x_k=1$, then the following constraints have to be satisfied:

$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$

and $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$.

→ If the above constraints are not satisfied then we will not proceed further in state space tree and we will backtrack and make $x_k=0$.

→ We will use two variables *s* and *r* as follows:

$s=\sum_{i=1}^{k} w_i x_i$

$r=\sum_{i=k+1}^{n} w_i$

Algorithm for Sum of Subsets problem using backtracking:

→ The algorithm is initially invoked as SumOfSub(0, 1, $\sum_{i=1}^{n} w_i$).

→ The solution vector $(x_1, x_2, \ldots, x_n)$ is initialized to zero.

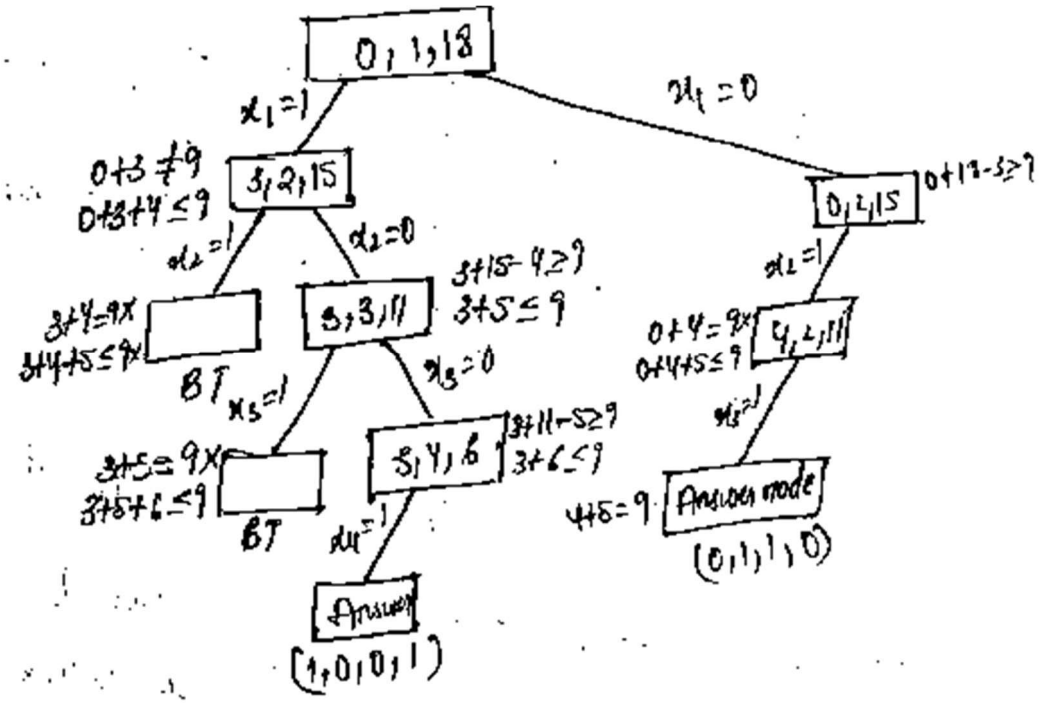
Algorithm SumOfSub(*s, k, r*)
// find all subsets of w[1: n] that sum to m.
// The value of x[j], 1≤j≤ k-1 have already been determined.
// At this point of time, s=$\sum_{j=1}^{k-1} w[j]x[j]$ *and r=*$\sum_{j=k}^{n} w[j]$.
// w[j]'s are in non-descending order.
// It's assumed that w[1]≤ m and $\sum_{i=1}^{n} w[i] \geq m$

```
{
        // generate left child
        x[k]:=1;
         if(s+w[k]=m) then          // subset found
                write(x[1: n]);
        else if(s+w[k]+w[k+1] ≤ m) then
                SumOfSub(s+w[k], k+1, r-w[k]);
        // otherwise generate right child.
        if((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)) then
        {
                x[k]:=0;
                SumOfSub(s, k+1, r-w[k]);
        }
}
```

Example:
Let *w= {3,4,5,6}* and *m=9*. Find all possible subsets of *w* that sum to *m*. Draw the portion of the state space tree that is generated.

Solution:
Note: In the state space tree, the rectangular nodes list the values of *s, k,* and *r* on each call to SumOfSub. Initially *s=0, k=1, r=18.*

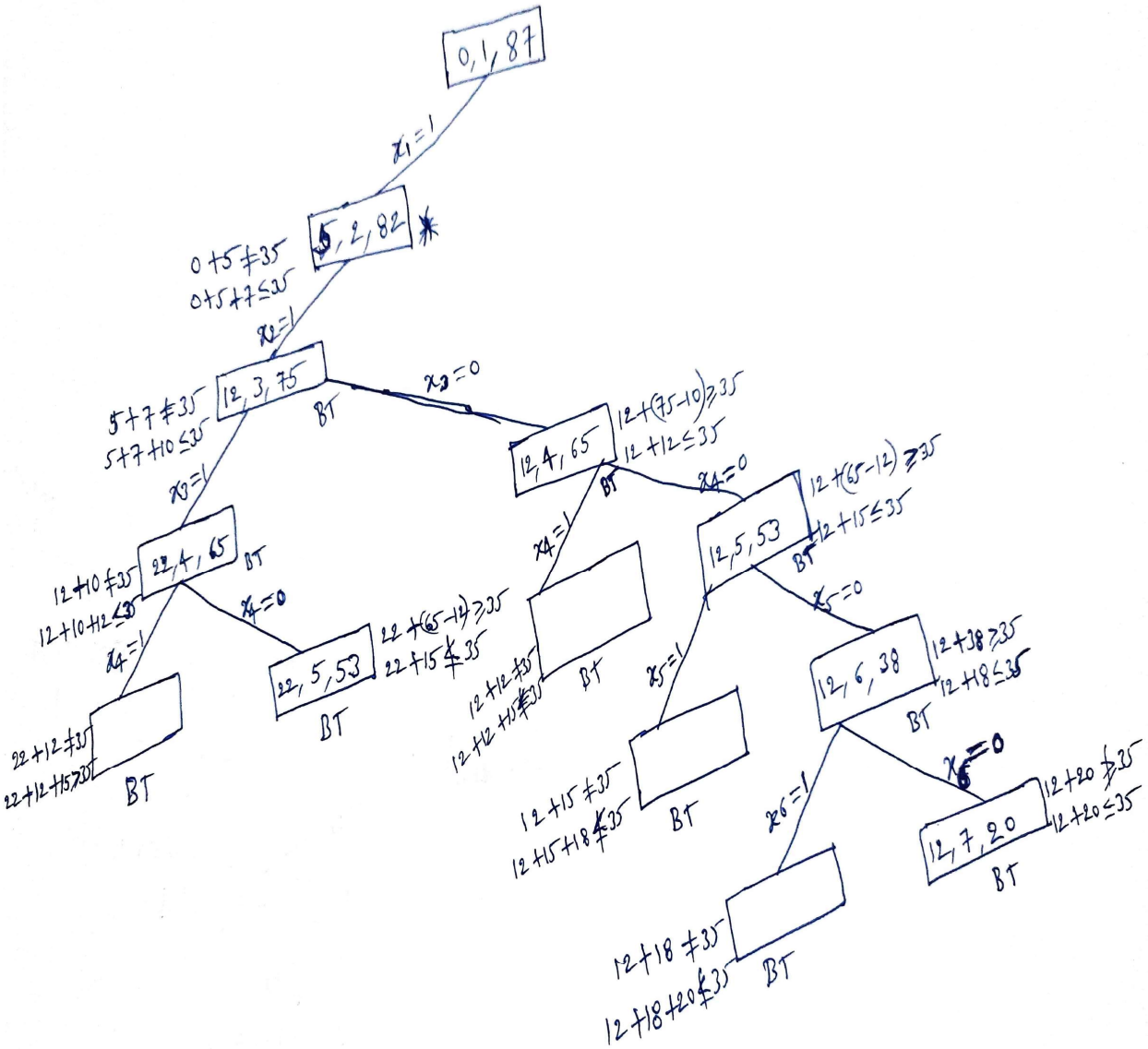
Portion of the state space tree that is generated:

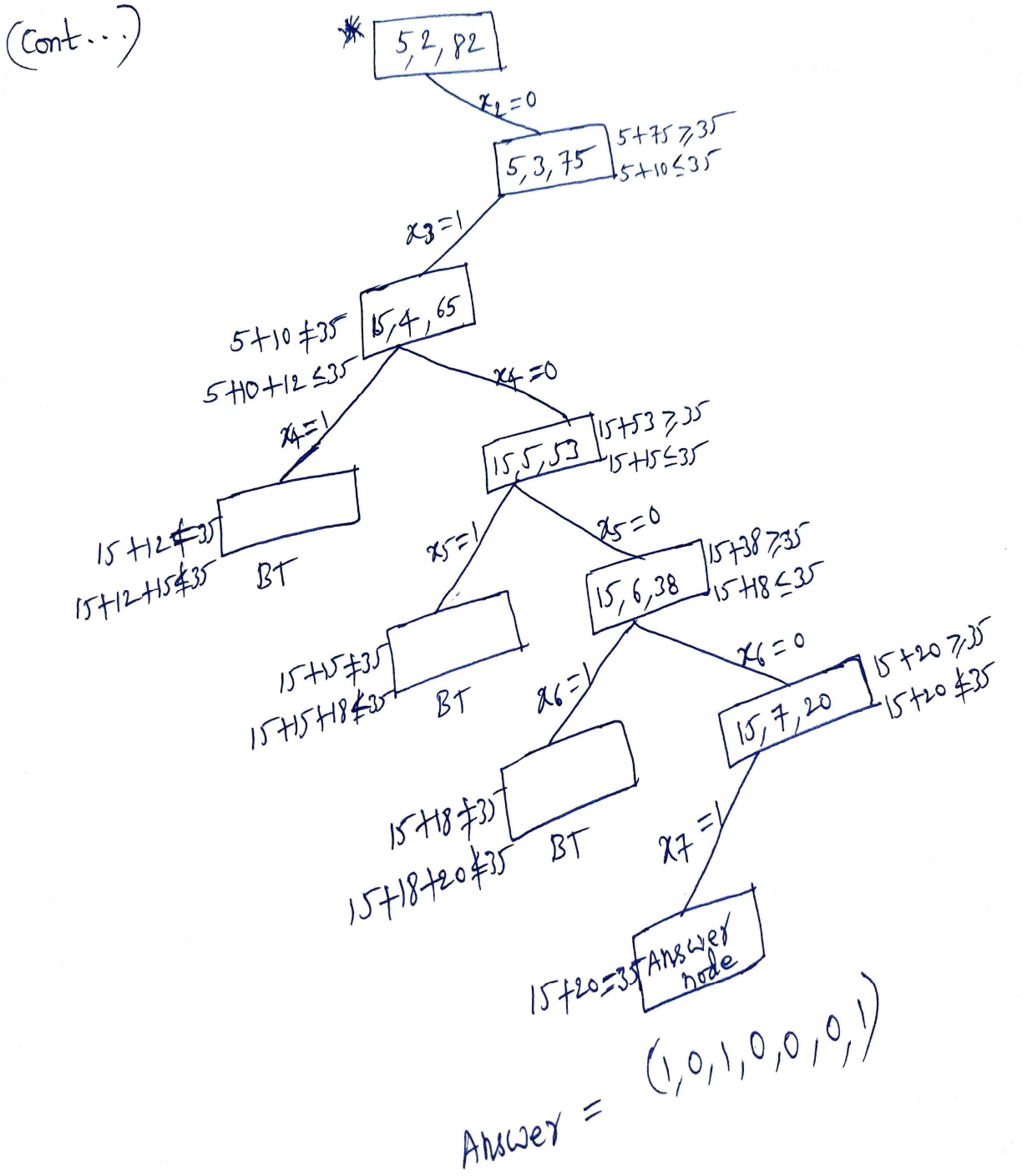


Example: **Let w = {5, 7, 10, 12, 15, 18, 20} and m=35. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.**

Solution:

Initially, sum of weights, r=5+7+10+12+15+18+20=87.


0, 1, 87
$x_1=1$
5, 2, 82 *
$0+5\neq 35$
$0+5+7\leq 35$
$x_2=1$
12, 3, 75
BT
$x_3=0$
$5+7\neq 35$
$5+7+10\leq 35$
$x_3=1$
12, 4, 65
$12+(75-10)\geqslant 35$
$12+12\leq 35$
22, 4, 65
BT
$x_4=1$
$x_4=0$
12, 5, 53
$12+(65-12)\geqslant 35$
$12+15\leq 35$
BT
$12+10\neq 35$
$12+10+12\leq 35$
$x_4=1$
$x_4=0$
22, 5, 53
$22+(65-12)\geqslant 35$
$22+15\neq 35$
BT
$22+12\neq 35$
$22+12+15\neq 35$
BT
$12+12\neq 35$
$12+12+15\neq 35$
BT
$x_5=1$
$x_5=0$
12, 6, 38
$12+38\geqslant 35$
$12+18\leq 35$
BT
$12+15\neq 35$
$12+15+18\neq 35$
BT
$x_6=1$
$x_6=0$
12, 7, 20
$12+20\neq 35$
$12+20\leq 35$
BT
$12+18\neq 35$
$12+18+20\neq 35$
BT



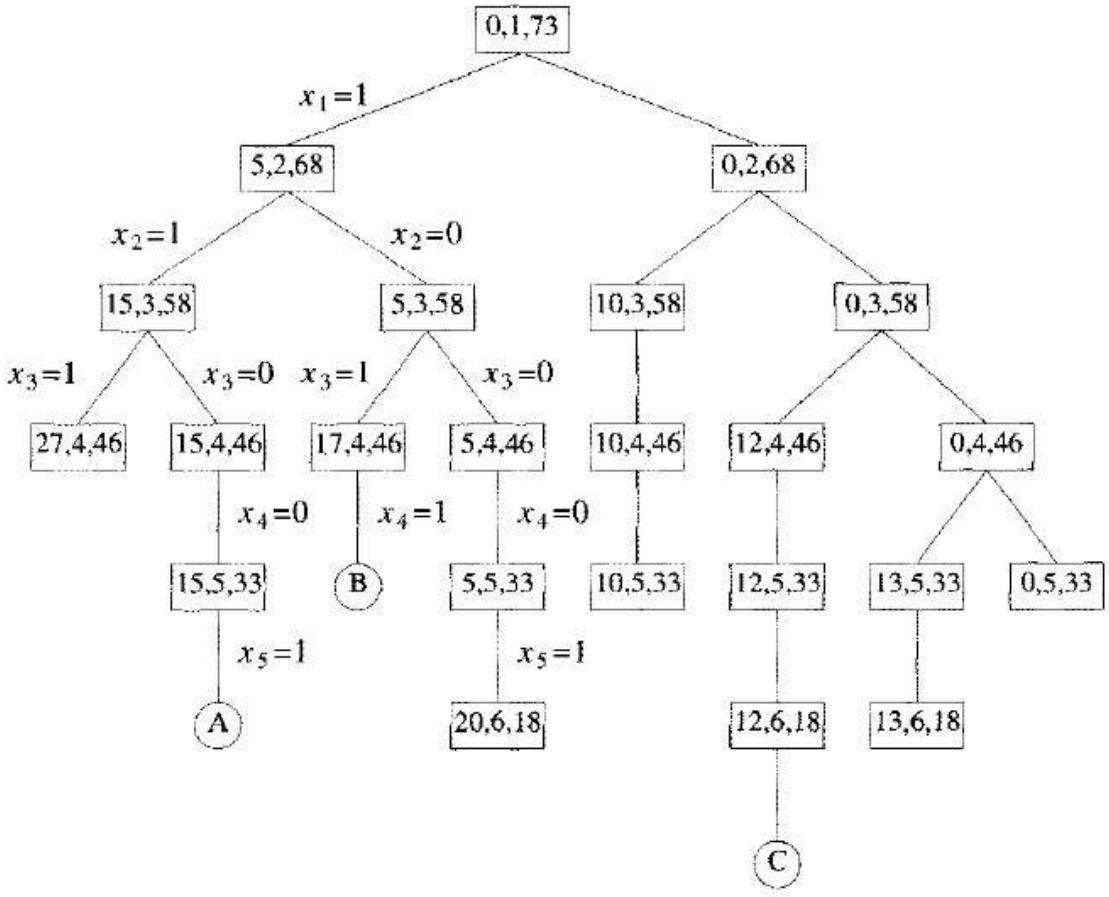
One solution is: (x1, x2, x3, x4, x5, x6, x7) = (1,0,1,0,0,0,1).

Example:

Let *w= {5,10,12,13,15}* and *m=30.* Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

Solution:

Note: In the state space tree, the rectangular nodes list the values of *s, k,* and *r* on each call to SumOfSub. Answer nodes are represented in circles. Initially *s=0, k=1, r=73.*



Branch and Bound

→ This technique is mostly used to solve optimization problems.

→ In branch and bound, a state space tree is constructed in such a way that all children of an E-node are generated before any other live node becomes an E-node.

→ Generated nodes that cannot possibly lead to a feasible solution are discarded. The remaining nodes are added to the list of live nodes, and then one node from this list is selected to become the next E-node. This expansion process continues until either the answer node is found or the list of live nodes becomes empty.

→ The next E-node can be selected in one of three ways:

1. FIFO (or) Breadth First Search: This scheme extracts nodes from the list of live nodes in the same order as they are placed in it. The live nodes list behaves as a queue.
2. LIFO (or) D Search: The live nodes list behaves as a stack.
3. LC Search (Least Cost Search) (or) Best First Search: The nodes are assigned ranks based on certain criteria and they are extracted in the order of Best-Rank-First.

→ Brach and Bound involves two iterative steps:

1. Branching:
   Splitting the problem into a number of subproblems.
   (or)
   Generating all the children of an E-node in the state space tree.

2. Bounding:
   Finding an ***optimistic estimate*** of the best solution to the subproblem.

Optimistic estimate: Upper bound for maximization problems.
Lower bound for minimization problems.

→ In case of LC Search Brach and Bound (LCBB) method, after generating the children of an E-node, the node with the best bound value (i.e., smallest lower bound in case of minimization problems or largest upper bound in case of maximization problems) is chosen from the list of all live nodes and is made the next E-node.

→ We terminate the search process at the current node of an LCBB algorithm because of any one of the following reasons:

1. The node represents an infeasible solution because constraints are not satisfied.

2. The bound value of the node is not better than the value of the best solution seen so far.

0/1 Knapsack problem:

Given *n* items with profits *(p1, p2, ..., pn)* and weights *(w1, w2, ..., wn)* and knapsack capacity *M*.

maximize $\sum_{i=1}^{n} p_i x_i$
Subject to the constraints
$\sum_{i=1}^{n} w_i x_i \leq M$
and
$x_i \in \{0,1\}$, $1 \leq i \leq n$.

Solution to the 0/1 Knapsack problem using LCBB:

→ 0/1 knapsack problem is maximization problem.

How to find the optimistic estimate (i.e., upper bound)?

We relax the integral constraint, i.e., $x_i \in \{0,1\}$, $1 \leq i \leq n$. That means, fractions of the items are allowed.

→ Arrange the items in the decreasing order of profit densities (*pi/wi* values).
→ In the state space tree, at every node we record three values, viz.,
W: Sum of the weights of the objects considered till now
P: Sum of the profits of the objects considered till now
UB: Upper bound on the optimal profit

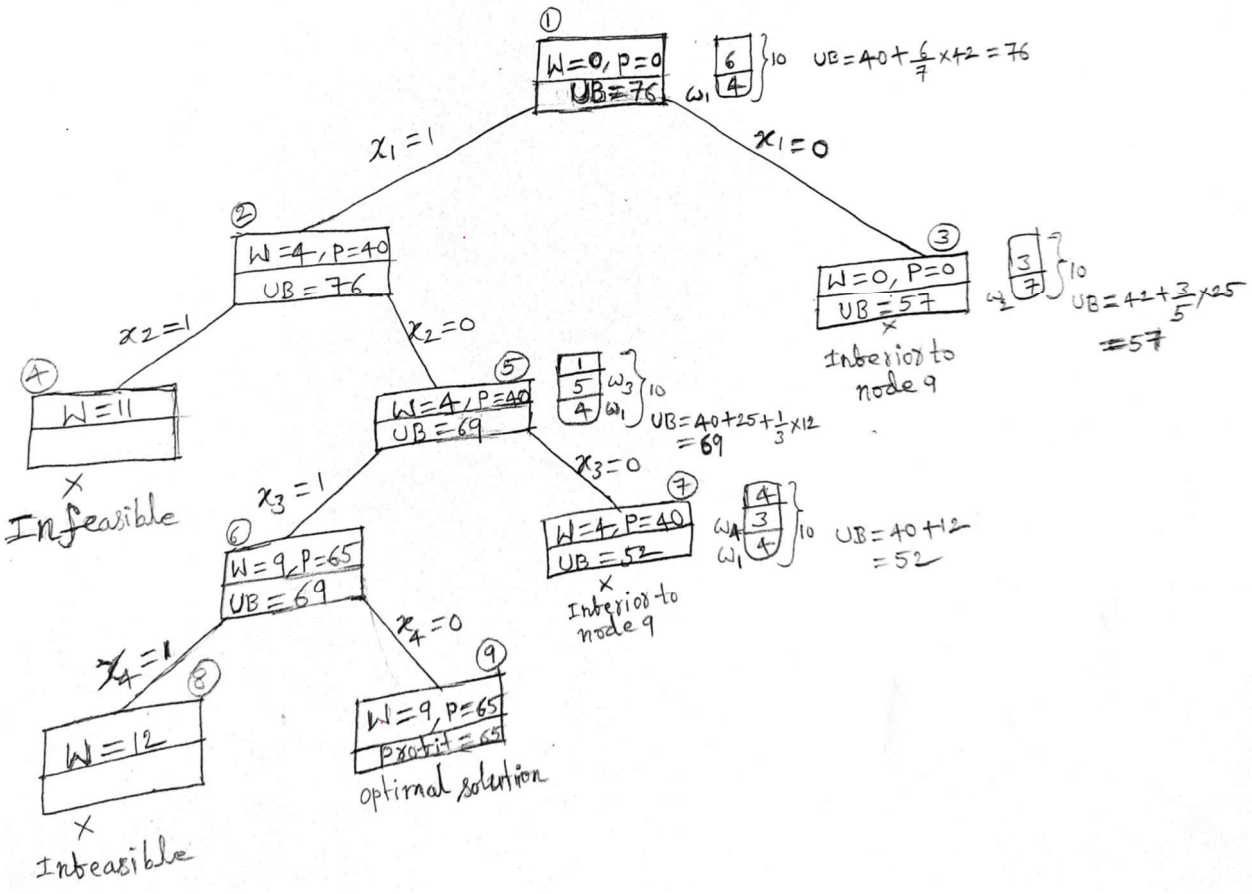
→ The upper bound is computed as follows:
Upper Bound = Sum of the profits of the items provided the total weight is less than or equal to knapsack capacity considering the fractions of items.

Example:

n=4; (w1, w2, w3, w4) = (4,7,5,3); (p1, p2, p3, p4) = (40,42,25,12) and *M=10.*

Solution:

i	*1*	*2*	*3*	*4*
Pi	*40*	*42*	*25*	*12*
wi	*4*	*7*	*5*	*3*
pi/wi	*10*	*6*	*5*	*4*



The optimal solution is: *(x1, x2, x3, x4) = (1,0,1,0)*

Exercise:

n=3 ; (w1, w2, w3) = (2,1,3) ; (p1, p2, p3,p4) = (10,4,6) and *M=5*.

UNIT-V

DETERMINISTIC ALGORITHMS:

The algorithms, in which the result (or, outcome) of every operation is uniquely defined, are called *deterministic algorithms.*

NON-DETERMINISTIC ALGORITHMS:

- The algorithms, in which the outcomes of certain operations may not be uniquely defined but are limited to the specified sets of possibilities (i.e., possible outcomes), are said to be *non-deterministic algorithms*.
- The theoretical (or, hypothetical) machine executing such operations is allowed to choose any one of these possible outcomes.

- The non-deterministic algorithm is a two-stage algorithm.

  1. Non-deterministic stage (or, Guessing stage):
  Generate an arbitrary string that can be thought of as a candidate solution to the problem.

  2. Deterministic stage (or, Verification stage):
  This stage takes the candidate solution and the problem instance as input and returns "yes" if the candidate solution represents actual solution.

- To specify non-deterministic algorithms, we use three functions:
  1. *Choice(S)*: arbitrarily chooses one of the elements of set 'S'.
  2. *Success( )*: signals a successful completion.
  3. *Failure( )*: signals an unsuccessful completion.

- The assignment statement x: = Choice(1, n) could result in x being assigned with any one of the integers in the range [1, n].

  There is no rule specifying how this choice is to be made. That's why the name *non-deterministic* came into picture.

- The Failure( ) and Success( ) signals are used to define a completion of the algorithm.

- Whenever there is a particular choice (or, set of choices (or) sequence of choices) that leads to a successful completion of the algorithm, then that choice (or, set of choices) is always made and the algorithm terminates successfully.

- *A nondeterministic algorithm terminates unsuccessfully, if and only if **there exists no set of choices** leading to a success signal.*

- The computing times for **Choice( ), Failure( ), Success( )** are taken to be O(1), i.e., constant time.

- A machine capable of executing a non-deterministic algorithm is called *non-deterministic machine.*

EXAMLE: 1: NON-DETERMINISTIC SEARCH:

```
Algorithm Nsearch(A, n, x)
{
    //A[1:n] is a set of elements, from which we have to determine
    //an index j, such that A[j]:=x, or 0 if x is not present in A.

    // Guessing Stage
    j := Choice(l, n);

    // Verification Stage
    if A[j] = x then
    {
        write(j);
        Success( );
    }
    write(0);
    Failure( );
}
```

➔ The time complexity is O(1)

EXAMLE 2: NON-DETERMINISTIC SORTING:

```
Algorithm NSort(A, n)
// A[1:n] is an array that stores n elements, which are positive
//integers.
// B[1:n] is an auxiliary array, in which elements are put at
//appropriate positions. That means, B stores the sorted elements.
{
      // guessing stage
      for i := 1 to n do
      {
            j := Choice(l, n);        //guessing the position of A[i] in B
            B[j] := A[i];             //place A[i] in B[j]
      }
      // verification stage
      for i:= 1 to n -1 do
      {
            if (B[i] > B[i+ 1]) then       // if not in sorted order.
                  Failure( );
      }
      Write(B[l : n]);                 // print sorted list.
      Success( );
}
```

Time complexity of the above algorithm is O(n).

➔ We mainly focus on *nondeterministic decision algorithms.*
Such algorithms produce either **'1'** or **'0'** (or, **Yes**/**No**) as their output.

→In these algorithms, a successful completion is made iff the output is 1. And, a 0 is output, iff there is no choice (or, sequence of choices) available leading to a successful completion.

→The output statement is implicit in the signals **Success( )** and **Failure( )**. No explicit output statements are permitted in a decision algorithm.

EXAMPLE: 0/1 KNAPSACK DECISION PROBLEM:

The knapsack decision problem is to determine if there is an assignment of 0/1 values to x_i, $1 \le i \le n$ such that $\sum_{i=1}^{n} p_i x_i \ge r$ and $\sum_{i=1}^{n} w_i x_i \le M$. r is a given number. The p_i's and w_i's nonnegative numbers.

```
1 Algorithm DKP(p, w, n, M, r, x)
2 {
3       W:= 0; P:= 0;
4       for i := 1 to n do
5       {
6               x[i]:= Choice(0, 1);
7               W := W + x[i] * w[i];
                P:=P+ x[i] * p[i];
8       }
9       if ((W>M) or (P < r)) then Failure( );
10      else Success( );
11 }
```

THE CLASSES P, NP, NP-HARD AND NP-COMPLETE:

→**P** is the set of all decision problems solvable by a deterministic algorithm in polynomial time.

→ An algorithm *A* is said to have *polynomial complexity (or, polynomial time complexity)* if there exists a polynomial *p( )* such that the computing time of *A* is *O(p(n))* for every input of size *n*.

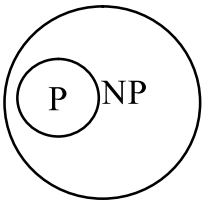
NP ***(Nondeterministic Polynomial time):***

→NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time.

→A non-deterministic machine can do everything that a deterministic machine can do and even more. This means that all problems in class P are also in class NP. So, we conclude that P ⊆NP.

→What we do not know, and perhaps what has become the most *famous unsolved problem* in computer science is, *whether P = NP or P ≠ NP*.

→The following figure displays the relationship between *P* and *NP* assuming that *P ≠NP*.



Some example problems in NP:

1. Satisfiability (SAT) Problem:

→SAT problem takes a Boolean formula as input, and asks whether there is an assignment of Boolean values (or, truth values) to the variables so that the formula evaluates to TRUE.

→A Boolean formula is a parenthesized expression that is formed from Boolean variables and Boolean operators such as OR, AND, NOT, IMPLIES, IF-AND-ONLY-IF.

→A Boolean formula is said to be in CNF (Conjunctive Normal Form, i.e., Product of Sums form) if it is formed as a collection of sub expressions called clauses that are combined using AND, with each clause formed as the OR of Boolean literals. A *literal* is either a variable or its negation.

→The following Boolean formula is in CNF:

$(x_1 \vee x_3 \vee x_5 \vee x_7) \wedge (x_3 \vee x_5) \wedge (x_6 \vee x_7)$

→The following formula is in DNF (Sum of Products form):

$(x_1 \wedge x_2 \wedge x_5) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$

→ *CNF-SAT* is the SAT problem for CNF formulas.

→It is easy to show that *SAT* is in *NP*, because, given a Boolean formula $E(x_1, x_2, \ldots, x_n)$, we can construct a polynomial time non-deterministic algorithm that could proceed by simply choosing (nondeterministically) one of the 2^n possible assignments of truth values to the variables $(x_1, x_2, \ldots, x_n)$ and verifying that the formula $E(x_1, x_2, \ldots, x_n)$ is **true** for that assignment..

Nondeterministic Algorithm for *SAT* problem:

```
Algorithm NSAT(E, n)
{
    //Determine whether the propositional formula E is satisfiable.
    //The variables are x1, x2, ..., xn.

    // guessing stage.
    for i:=1 to n do          // Choose a truth value assignment.
        xi := Choice(false, true);

    // verification stage.
     if E(x1, x2, ..., xn) = true then Success( );
    else Failure( );
}
```

→Time complexity is O(n), which is a polynomial time. So, *SAT* is NP problem.

2. CLIQUE PROBLEM:

Clique: A *clique* of a graph 'G' is a complete subgraph of G.
→The size of the clique is the number of vertices in it.

Clique problem: Clique problem takes a graph 'G' and an integer 'k' as input, and asks whether G has a clique of size at least 'k'.

Nondeterministic Algorithm for Clique Problem:

```
Algorithm DCK(G, n, k)
{
    //The algorithm begins by trying to form a set of k distinct
    //vertices. Then it tests to see whether these vertices form a
    //complete sub graph.

    // guessing stage.
     S := Ø;     // S is an initially empty set.
    for i := 1 to k do
    {
        t := Choice(l, n);
        S := S U {t}       // Add t to set S.
    }
    //At this point, S contains k distinct vertex indices.
    //Verification stage
    for all pairs (i, j) such that i ∈ S, j ∈ S, and i ≠ j do
        if (i, j) is not an edge of G then Failure( );
    Success( );
}
```

→A nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time complexity of its verification stage is polynomial.

→**Tractable Problems:** Problems that can be solved in polynomial time are called ***tractable***.
Intractable Problems: Problems that cannot be solved in polynomial time are called ***intractable***.

→Some decision problems cannot be solved at all by any algorithm. Such problems are called ***undecidable***, as opposed to ***decidable*** problems that can be solved by an algorithm.
→A famous *example of an undecidable* problem was given by Alan Turing in 1936. It is called the ***halting problem***: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

REDUCIBILITY:

→ A decision problem D_1 is said to be ***polynomially reducible*** to a decision problem D_2 *(also written as* $D_1 \propto D_2$*)*, if there exists a function *t* that transforms instances of D_1 into instances of D_2 such that:

1. *t* maps all *Yes* instances of D_1 to *Yes* instances of D_2 and all *No* instances of D_1 to *No* instances of D_2.

2. *t* is computable by a polynomial time algorithm.

→The definition for $D_1 \propto D_2$ immediately implies that if D_2 can be solved in *polynomial time*, then D_1 can also be solved in *polynomial time*. In other words, if D_2 has a deterministic polynomial time algorithm, then D_1 can also have a deterministic polynomial time algorithm.

Based on this, we can also say that, if D_2 is easy, then D_1 can also be easy. In other words, D_1 is as easy as D_2. Easiness of D_2 proves the easiness of D_1.

→But, here we mostly focus on showing *how hard a problem is* rather than how easy it is, by using the contra positive meaning of the reduction as follows:

$D_1 \propto D_2$ implies that if D_1 cannot be solved in *polynomial time*, then D_2 also cannot be solved in *polynomial time*. In other words, if D_1 does not have a deterministic polynomial time algorithm, then D_2 also can not have a deterministic polynomial time algorithm.

We can also say that, if D_1 is hard, then D_2 can also be hard. In other words, D_2 is as hard as D_1.

→To show that problem D_1 (i.e., new problem) is at least as hard as problem D_2 (i.e., known problem), we need to reduce D_2 to D_1 (not D_1 to D_2).

→Reducibility ($\propto$) is a transitive relation, that is, if $D_1 \propto D_2$ and $D_2 \propto D_3$ then $D_1 \propto D_3$.

NP-HARD CLASS:

→A problem 'L' is said to be NP-Hard iff every problem in NP reduces to 'L'

(or)

→ A problem 'L' is said to be NP-Hard if it is as hard as any problem in NP.

(or)

→A problem 'L' is said to be NP-Hard iff SAT reduces to 'L'.

Since SAT is a known NP-Hard problem, every problem in NP can be reduced to SAT. So, if SAT reduces to L, then every problem in NP can be reduced to 'L'.

Ex: SAT and Clique problems.

→An NP-Hard problem *need not be* NP problem.

Ex: *Halting Problem* is NP-Hard **but not** NP.

NP-COMPLETE CLASS:

→A problem 'L' is said to be NP-Complete if 'L' is NP-Hard and L ∈ *NP*.

→These are the hardest problems in NP set.

Ex: SAT and Clique problems.

Showing that a decision problem is *NP*-complete:

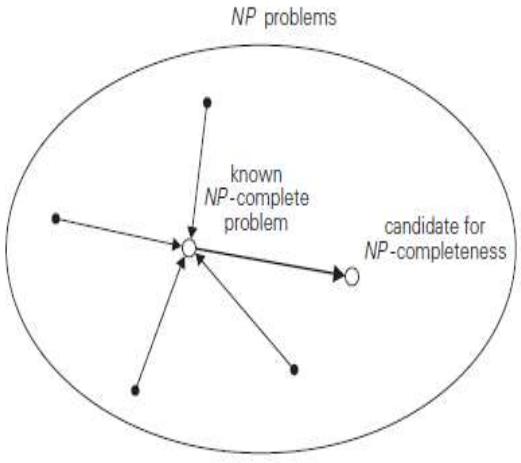
It can be done in two steps:

Step1:

Show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy.

Step2:

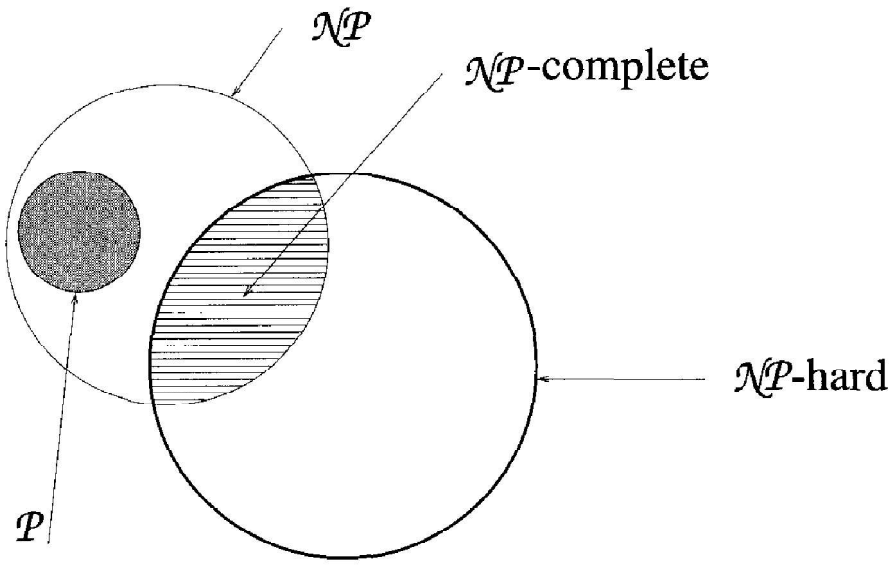
Show that the problem in question is NP-Hard also. That means, show that every problem in *NP* is reducible to the problem in question, in polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known *NP*-complete problem can be transformed into the problem in question, in polynomial time, as depicted in the figure below.



Proving *NP*-completeness by reduction.

→The definition of *NP*-completeness immediately implies that if there exists a polynomial-time algorithm for just one *NP*-Complete problem, then every problem in *NP* can also have a polynomial time algorithm, and hence *P* = *NP*.

Relationship among P, NP, NP-Hard and NP-Complete Classes:



COOK'S THEOREM:

→Cook's theorem can be stated as follows.

(1) SAT is NP-Complete.

(or)

(2) If *SAT* is in *P* then *P* = *NP*. That means, if there is a polynomial time algorithm for *SAT*, then there is a polynomial time algorithm for every other problem in *NP*.

(or)

(3) *SAT* is in *P* iff *P* = *NP*.

Application of Cook's Theorem:

A new problem 'L' can be proved NP-Complete by reducing SAT to 'L' in polynomial time, provided 'L' is NP problem. Since SAT is

NP-Complete, every problem in NP can be reduced to SAT. So, once SAT reduces to ‘L’, then every problem in NP can be reduced to ‘L’ proving that ‘L’ is NP-Hard. Since ‘L’ is NP also, we can say that ‘L’ is NP-Complete.

Example Problem: Prove that Clique problem is NP-Complete.

(OR)

Reduce SAT problem to Clique problem.

Solution: See the video at https://www.youtube.com/watch?v=qZs767KQcvE
