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 4th 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: $<, >, \leq, \geq, =, \neq$

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.

9. 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**.

10. 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>
}
```

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

12. 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 ith element is stored in the ith position, i.e., a[i] , $1 \leq i \leq n$.
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;
        }
    }
}
```

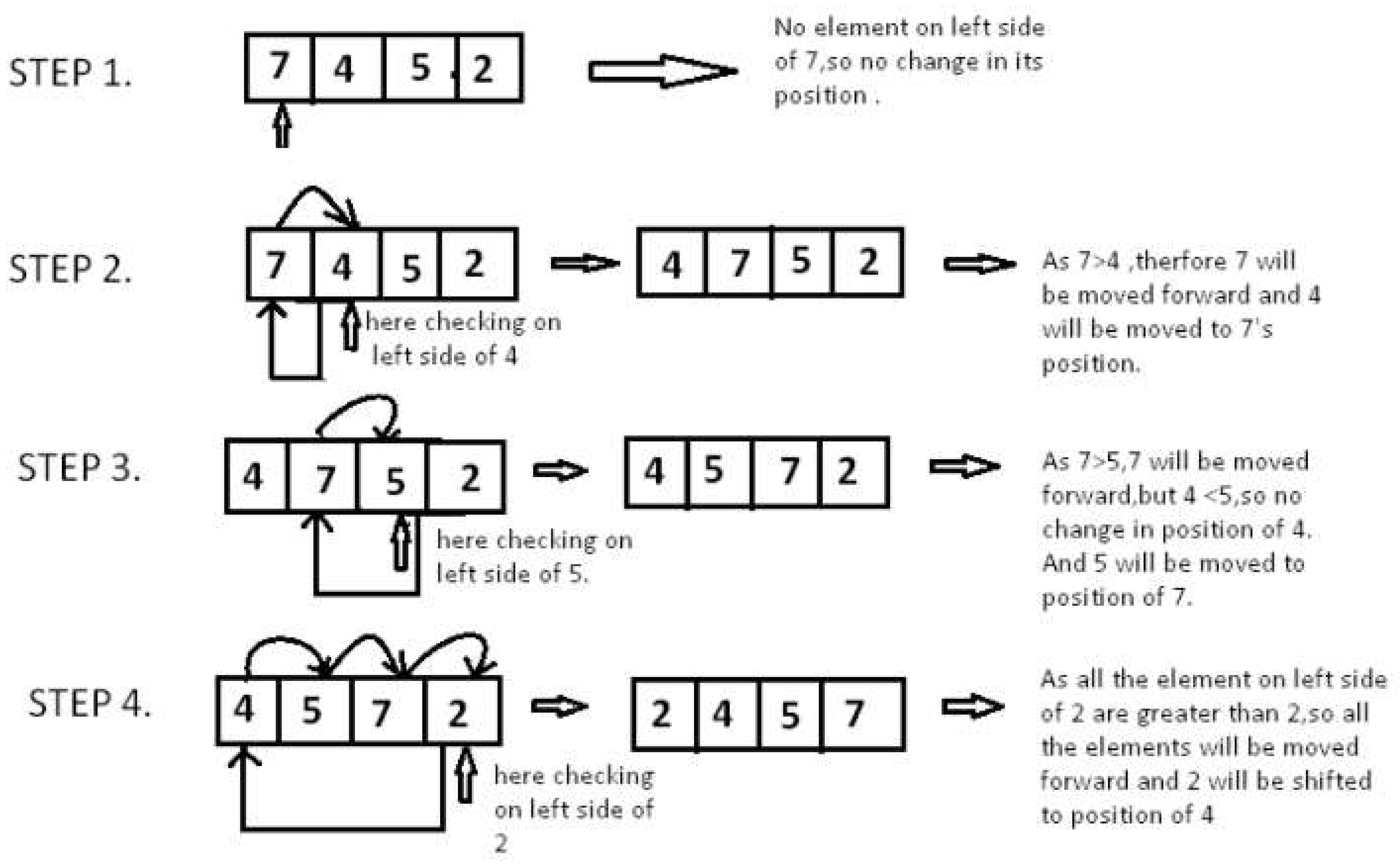
Algorithm for Insertion Sort:

```
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 $A[] = [7, 4, 5, 2]$.


STEP 1.
7 4 5 2
No element on left side of 7,so no change in its position .
STEP 2.
7 4 5 2
here checking on left side of 4
4 7 5 2
As 7>4 ,therfore 7 will be moved forward and 4 will be moved to 7's position.
STEP 3.
4 7 5 2
here checking on left side of 5.
4 5 7 2
As 7>5,7 will be moved forward,but 4 <5,so no change in position of 4. And 5 will be moved to position of 7.
STEP 4.
4 5 7 2
here checking on left side of 2
2 4 5 7
As all the element on left side of 2 are greater than 2,so all the elements will be moved forward and 2 will be shifted to position of 4

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.

<u>EXAMPLE-1</u>:-

```
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.

<u>EXAMPLE 2:</u> 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.

<u>EXAMPLE-3</u>:-

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).

<u>EXAMPLE-4</u>:-

```
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).

<u>Time complexity</u>:-

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 .

<u>Determining the time complexity by step count method:</u>

→The time complexity of an algorithm is given by the <u>number of steps</u> 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;	1	1	1
for i:=1 to n do	n+1	1	n+1
s:=s+a[i];	n	1	n
return s;	1	1	1
}			
		Total step count of algorithm =	2n+3

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

The step count tells us how the runtime for a program changes 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**	1	1	1	1	1
return 0.0;	1	0	1	1	0
else					
return Rsum (a,n-1)+ a(n);	0	1	$1+t_{Rsum}(n-1)$	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

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

<u>EXAMPLE-3:-</u> 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**	m+1	1	m+1
for j:=1 to n **do**	m(n+1)	1	mn+ m
c[i,j]:=a[i,j]+ b[i,j];	mn	1	mn
}			
		Total step count =	2mn+2m+1

<u>EXAMPLE-3a:-</u> 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**	n+1	1	n+1
for j:=1 to n **do**	n(n+1)	1	n^2+ n
c[i,j]:=a[i,j]+ b[i,j];	n^2	1	n^2
}			
		Total step count =	$2n^2+2n+1$

<u>EXAMPLE-3b:-</u> 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**	n+1	1	n+1
{			
for j:=1 to n **do**	n(n+1)	1	n^2+ n
{			
c[i,j]:=0;	n^2	1	n^2
for k:=1 to n **do**	n^2(n+1)	1	n^3+ n^2
{			
c[i,j]:= c[i,j]+a[i,k]* b[k,j];	n^3	1	n^3
}			
}			
}			
}			
		Total step count =	$2n^3+3n^2+2n+1$

<u>EXAMPLE-4</u>:- 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_l and second term as f_2, then f1=0 and f2=1, and in general,

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

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**	1	1	1	1	1
write(n-1);	1	0	1	1	0
else					
{					
fn-2:=0; fn-1:=1;	0	1	2	0	2
for i:=3 to n **do**	0	n-1	1	0	n-1
{					
fn:=fn-1+fn-2;	0	n-2	1	0	n-2
fn-2:=fn-1;	0	n-2	1	0	n-2
fn-1:=fn;	0	n-2	1	0	n-2
}					
write(fn);	0	1	1	0	1
}					
}					
			Total step count =	2	4n-3

<u>Order (or, Rate) of growth of running time:</u>
→Our motivation to determine step count is <u>to compare the running times of two alternative algorithms that perform the same task</u> and also <u>to predict how the running time of an algorithm grows as its input size changes</u>.

So, would like to determine the <u>order of growth of the running time</u> 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.

<u>EXAMPLE:-</u>
→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.

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

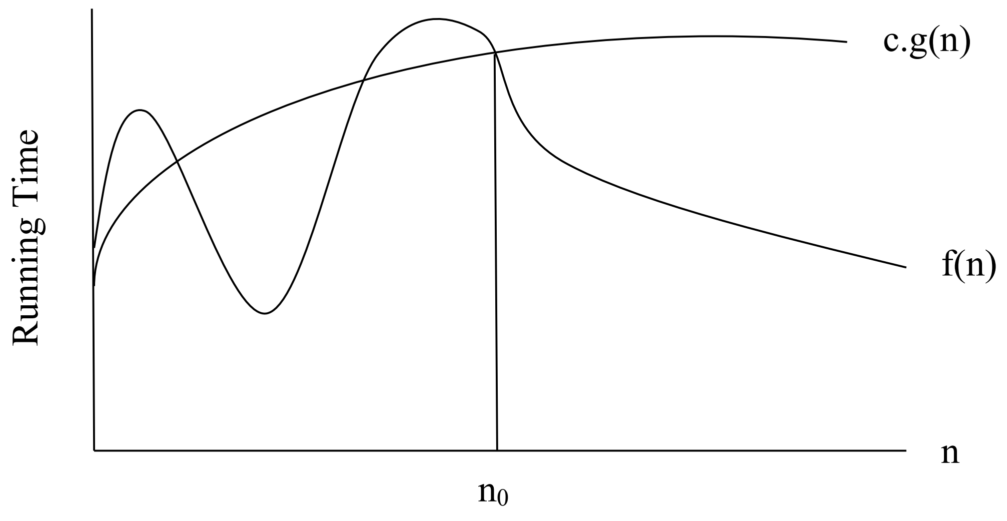
However, asymptotic notations actually apply to functions.

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 <u>how fast this runtime function grows with the input size</u> 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 $f(n) \le cg(n)$ for all $n \ge n_0$.



Ex:-

1) 3n+2=O(n)

Here,

f(n)=3n+2

g(n)=n.

3n+2=O(n) as $3n+2\le 4n$ for all $n\ge2$, where c=4 and $n_0=2$.

2) 3n+3=O(n)

3) 100n+6=O(n), as $100n+6\le 101n$, for all $n\ge6$.

4) $10n^2+4n+2=O(n^2)$, as $10n^2+4n+2\le 11n^2$, for all $n\ge5$. (or) $10n^2+4n+2\le 16n^2$, for all $n\ge1$.

5) $6*2^n+n^2=O(2^n)$, as $6*2^n+n^2\le 7*2^n$, for all $n\ge4$.

6) $3n+3=O(n^2)$, as $3n+3\le 3n^2$, for all $n\ge1$ **(or)** $3n+3\le n^2$, for all $n\ge4$

7) $2^{100}=O(1)$, as $2^{100}\le 2^{100}.1$, for all $n\ge1$.

Note:-We write O(1) to mean a computing time of constant.

Theorem:- If $f(n)=a_mn^m+.........+a_1n+a_0$, then $f(n) = O(n^m)$.

Proof:- $f(n) = \sum_{i=0}^{m} a_in^i$

$\le \sum_{i=0}^{m} |a_i|n^i = n^m\sum_{i=0}^{m} |a_i|n^{i-m}$

$\le n^m\sum_{i=0}^{m} |a_i|$, for all $n\ge1$.

$\le c.g(n)$, where $c=\sum_{i=0}^{m} |a_i|$ and $g(n)= n^m$.

So, $f(n) = O(n^m)$.

Ex:- $10n^2+4n+2=O(n^2)$ as $10n^2+4n+2\le16n^2$ for all $n\ge1$.

→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 $n\ge n_0$. 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:- $n^2+nlogn+n+4=O(n^2)$.

$3n+2\ne O(1)$, as 3n+2 is not less than or equal to any constant c for all $n\ge n_0$.

$10n^2+4n+2\ne O(n)$.

Problem1: (GATE-2017 Set1)

Consider the following functions from positives integers to real numbers:

10, √n, n, $\log_2 n$, 100/n.

The CORRECT arrangement of the above functions in increasing order of asymptotic complexity is:

(A) $\log_2 n$, 100/n, 10, √n, n

(B) 100/n, 10, $\log_2 n$, √n, n

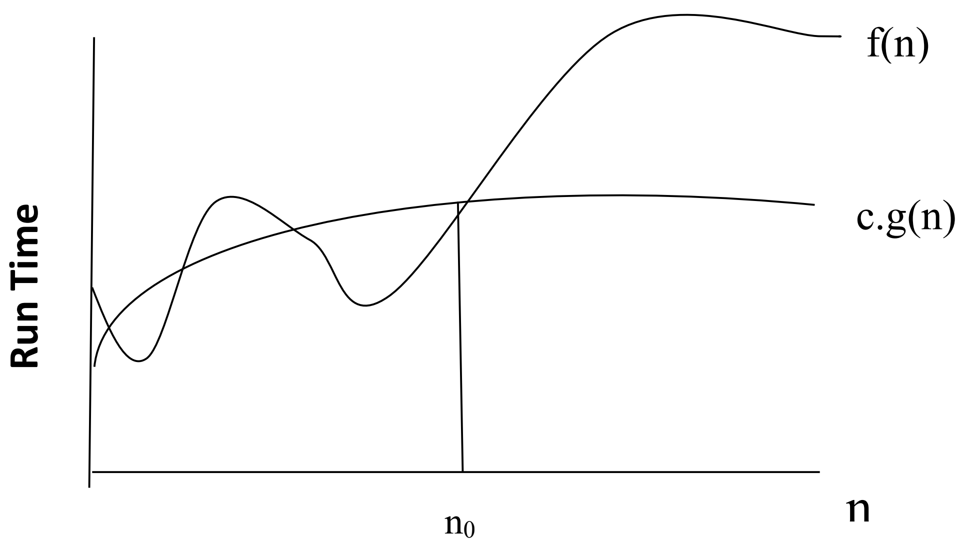
(C) 10, 100/n ,√n, $\log_2 n$, n

(D) 100/n, $\log_2 n$, 10 ,√n, n

NOTE:-
1)$n!=O(n^n)$
2)$\log n^3=O(\log n)$.

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 $\Omega(g(n))$ 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 $n\geq n_0$.

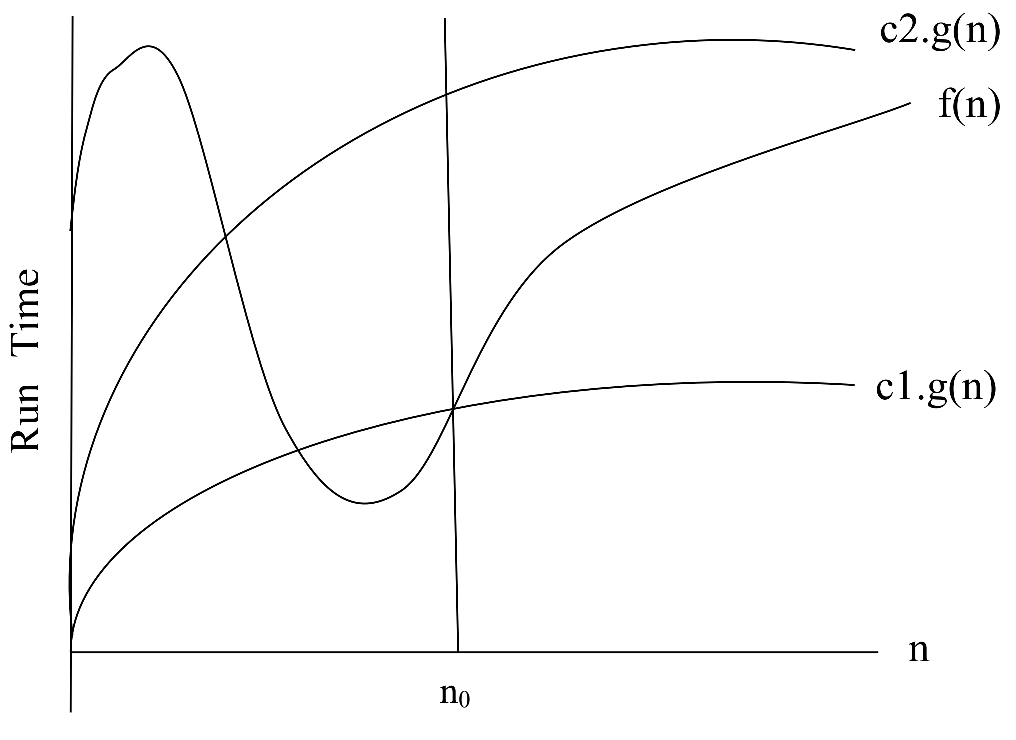


Ex:-
1)$3n+2=\Omega(n)$as, $3n+2\geq 3n$ for all $n\geq 1$ where c=3 and $n_0=1$.
2)$3n+3=\Omega(n)$ as,$3n+3\geq 3n$ for all $n\geq 1$.
3)$100n+6=\Omega(n)$ as, $100n+6\geq 100n$ for all $n\geq 1$.
4)$10n^2+4n+2=\Omega(n^2)$ as, $10n^2+4n+2\geq n^2$ for all $n\geq 1$, where c=1 and $n_0=1$.
5)$6*2^n+n^2=\Omega(2^n)$ 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 $\Theta(g(n))$ 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, $f(n)=\Theta(g(n))$ iff $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$.
We can say that $f(n)=\Theta(g(n))$ 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)$3n+3=\Theta(n)$ since $3n+3=O(n)=\Omega(n)$
3)$10n^2+4n+2=\Theta(n^2)$.
4)$6*2^n+n^2=\Theta(2^n)$.
5)$10*\log n+4=\Theta(\log n)$.
6)$3n+2\neq\Theta(1)$ since $3n+2=\Omega(1)$ but $3n+2\neq O(1)$

7)3n+3$\neq\Theta(n^2)$.
8)$10n^2+4n+2\neq\Theta(n)$ and
$10n^2+4n+2\neq\Theta(1)$.
9)$6*2^n+n^2\neq\Theta(n^2)$ and $6*2^n+n^2\neq\Theta(1)$.

<u>Little-oh naotation(o):-</u> (f(n)<g(n))
The function f(n)=o(g(n)) iff $\lim_{n\to\infty} f(n)/g(n)$=0.
<u>Ex:-</u>
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 $\lim_{n\to\infty}(3n+2)/n=\lim_{n\to\infty}3+(\frac{2}{n})=3\neq 0$.

<u>Little omega notation(ω):-</u> (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) $10n^2+4n+2=\omega(n)$ since $\lim_{n\to\infty} n/(10n^2+4n+2)=0$.
$=\omega(1)$
$\neq\omega(n^2)$

<u>Problem:</u> - (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

<u>Hint:</u>
$$\sum_{1\le i\le n} i^k = \frac{n^{k+1}}{k+1}+\frac{n^k}{2}+\text{lower order terms}$$

<u>Problem:</u> - (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.

<u>Asymptotic Notations and Time Complexities of Algorithms:</u>

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
$k(k+1)/2=n \Rightarrow k=O(\sqrt{n})$.

<u>Practice Problem:</u>

What is the complexity of the following code?

```
sum=0;
   for(i=1;i<=n;i*=2)
       for(j=1;j<=n;j++)
         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) $n\log n$
(D) $n\log(\log n)$

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**	1	n
if(a[i]=x) **then**	1	n
return i;	1	1
return 0;	0	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}.

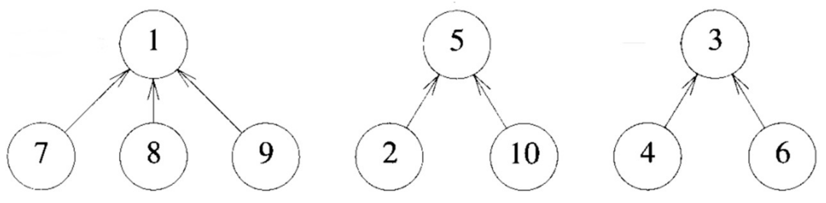
→The following two operations are performed on the disjoint sets:
1) Union:
If $\mathbf{S_i}$ and $\mathbf{S_j}$ are two disjoint sets, then their union $\mathbf{S_i}$ U $\mathbf{S_j}$ = {all elements **x** such that **x** is in $\mathbf{S_i}$ or $\mathbf{S_j}$}. Thus, in our example, $\mathbf{S_1}$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

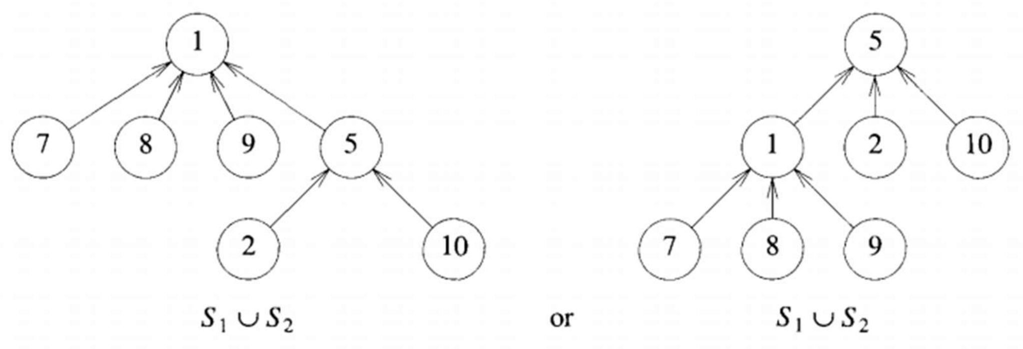
In presenting the UNION and FIND algorithms, we ignore the set names and identify sets just by the roots of the trees representing sets. This simplifies the discussion.

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 U S_2 :

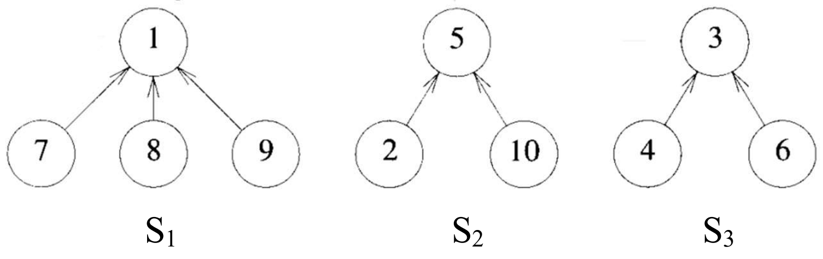


Representing the tree nodes of all disjoint sets using a single array:
Since the universal set elements are numbered 1 through n, we represent the tree nodes of all sets using an array P[1:n], where P stands for parent.

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)**

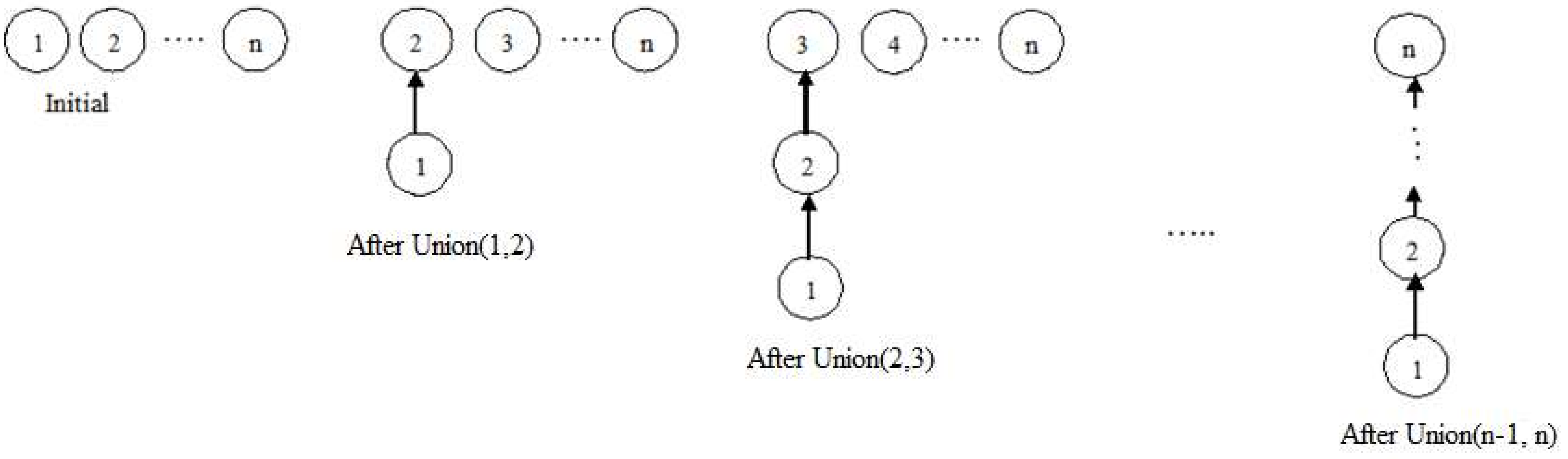
```
Algorithm SimpleUnion(i, j)
{
    P[i] := j;
}
```

→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:

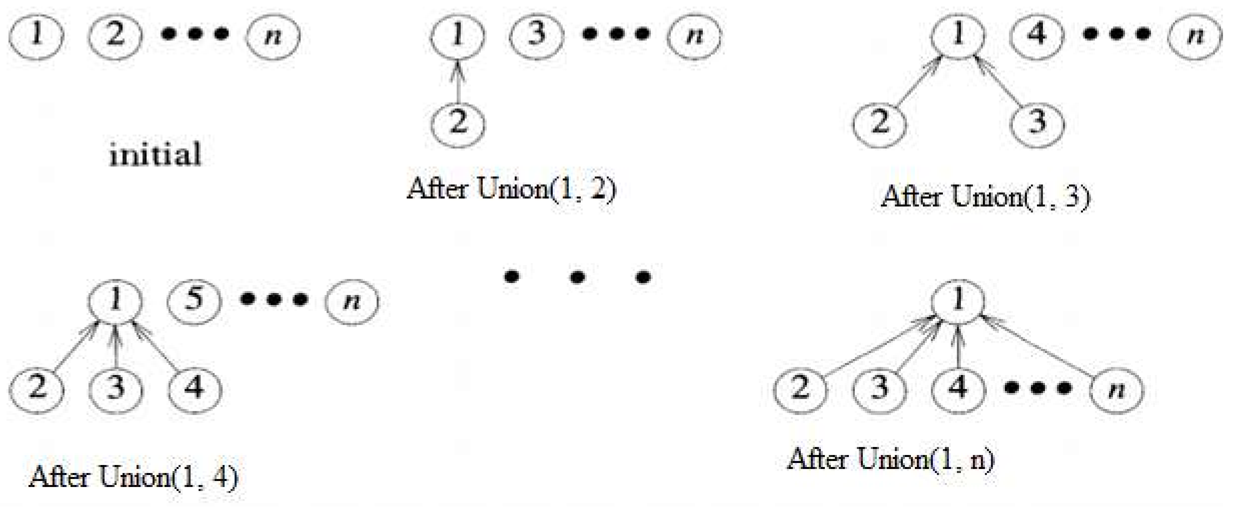
Find(1), Find(2), …, Find(n).

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 $\sum_{i=1}^{n} i$ **= O(n²).**

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 :

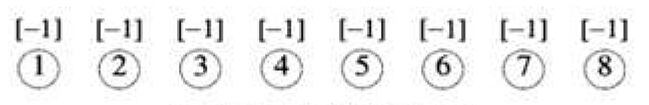
```
Algorithm WeightedUnion(i, j)
{
    // Unite sets with the roots i and j (i ≠ j) using weighting rule.
    // P[i] = -count[ i ] and P[ j ] = -count[ j ]
    temp := P[ i ] + P[ j ];
    if(P[ i ] > P[ j ]) then    // if tree 'i' has lesser number of nodes than tree 'j'
    {
        P[i ] := j;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

    }
}
```

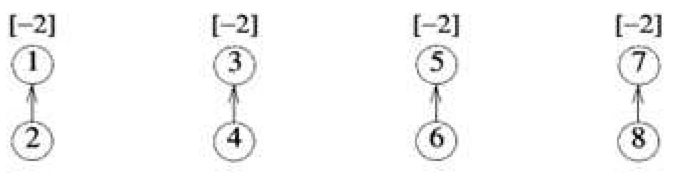
The time taken for WeightedUnion(i, j) is also constant, that is, **O(1)**.

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)**.

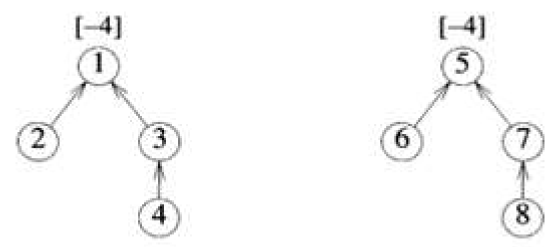
Example:
Consider the behavior of WeightedUnion on the following sequence of UNION operations starting from the initial configuration, P[i] = - count[i] = -1, $1 \le i \le 8$:
Union(1,2), Union(3,4), Union(5,6), Union(7,8), Union(1,3), Union(5,7), Union(1,5):



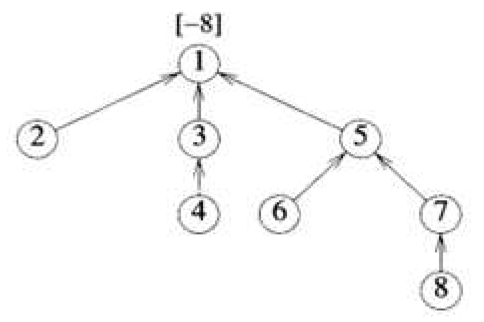
(a) Initial height-0 trees



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



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



(d) Height-3tree 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.

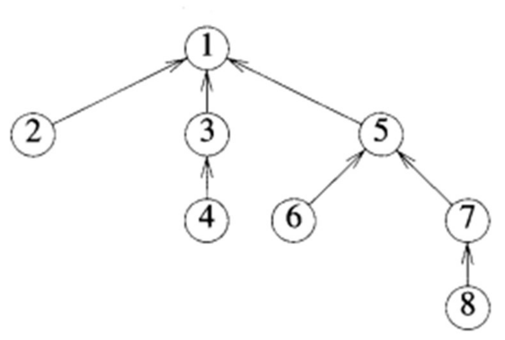
Collapsing Rule :

If 'j' is a node on the path from 'i' to its root 'r' and P[i] $\neq$ 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:

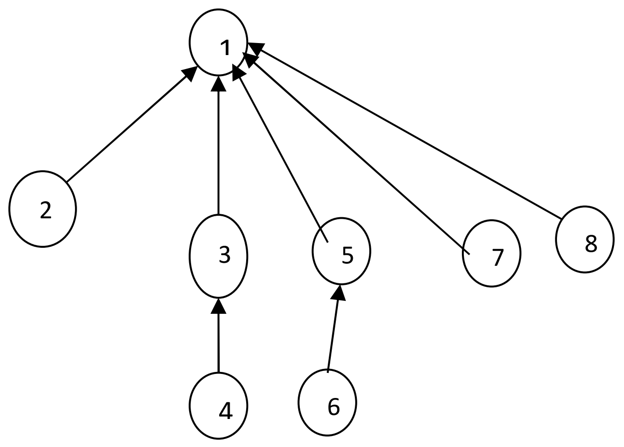


i	1	2	3	4	5	6	7	8
P [i]	-8	1	1	3	1	5	5	7

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:

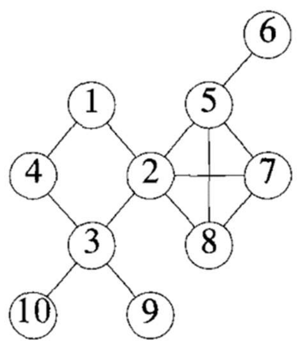


Each of the remaining seven Find(8) operations require going up only one parent link field. The total cost is now only 3+3+7=13 moves.

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:

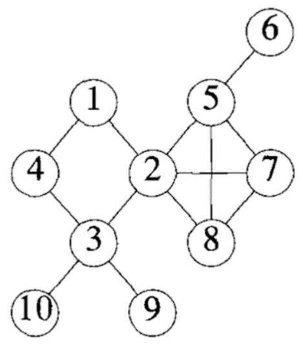


The articulation points in the graph G are: 2, 3 and 5.

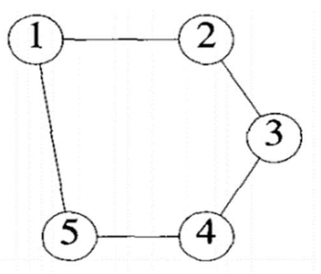
Biconnected Graph: A graph G is biconnected if and only if it contains no articulation points.

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.

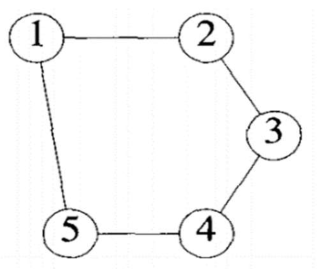
On the other hand, if G has no articulation point, then if any station i fails, we can still communicate between any two stations excluding station i.

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.

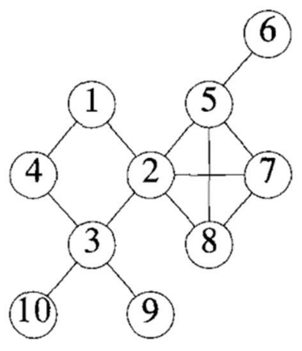
Ex: Consider the following biconnected graph:



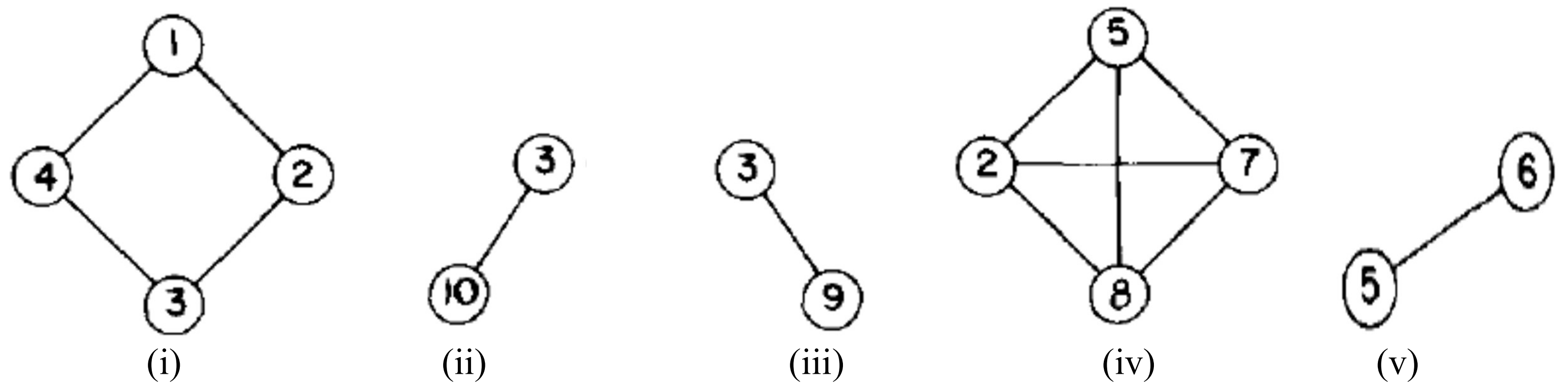
This graph has only one biconnected component (i.e., the entire graph itself).

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:


(i) (ii) (iii) (iv) (v)

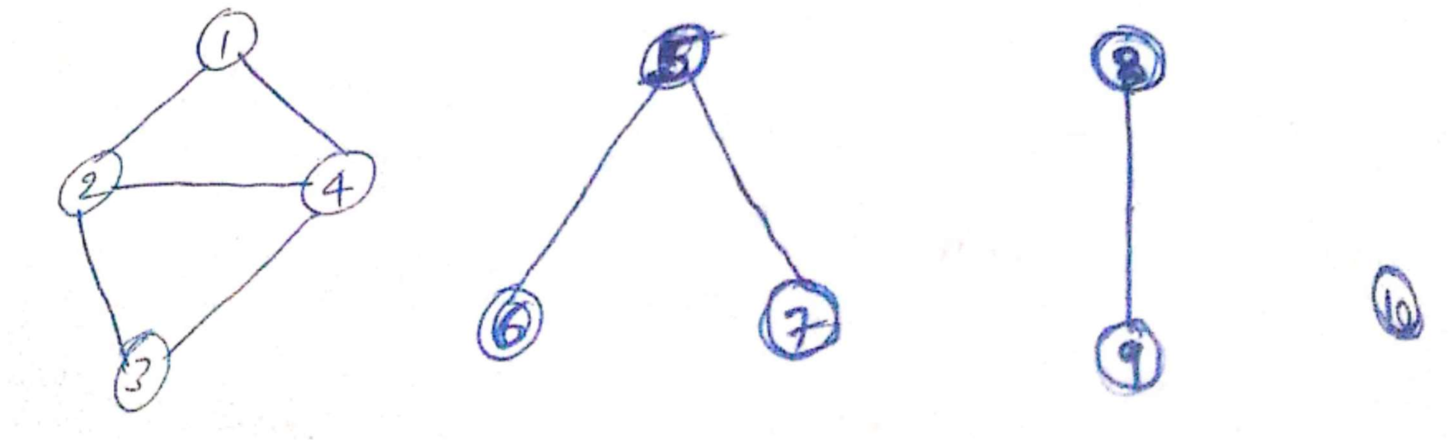
Note: Two biconnected components can have at most one vertex in common and this vertex is an articulation point.

Connected Components:

A connected component of a graph G is a maximal subgrph 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:



Amortized Analysis

In an amortized analysis, we average the time required to perform a sequence of data-structure operations.
With amortized analysis, we can show that the average cost of any operation in the sequence is cheap although a single operation in the sequence might be expensive.

Amortized analysis applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure where the costs of those operations vary over a period of time.
It turns out that in some situations a single operation can be expensive, but the total time for an entire sequence of *n* such operations is always significantly lesser than the worst-case time complexity of that single operation multiplied by *n*.

There are three most common techniques used in amortized analysis:
Aggregate analysis
Accounting method
Potential method

(1) Aggregate analysis:
We determine an upper bound T(*n*) on the total cost of a sequence of *n* operations. The average cost per operation is then T(*n*)/*n*. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

$$\boxed{\text{Amortized cost of each operation} = \frac{Sum\ of\ the\ actual\ costs\ of\ n\ operations\ in\ a\ sequence}{n}}$$

(2) Accounting method:
The accounting method is aptly named because it borrows ideas and terms from accounting.

Here, each operation is charged a cost called the **amortized cost**. Some operations can be charged more or less than they actually cost. If an operation's amortized cost exceeds its actual cost, the surplus is added to a **credit** (which is like a bank balance). Credit can be used later to help pay for other operations whose amortized cost is less than their actual cost. Total credit can never be negative in any sequence of operations.

We must choose the amortized cost of each operation carefully.

Different types of operations (like *push* and *pop* in stack data structure or *insert* and *delete* in queue data structure) may have different amortized costs. This method differs from aggregate analysis wherein all types of operations have the same amortized cost.

We must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence of operations. That means, the total amortized cost of a sequence of operations must be greater than or equal to the total actual cost of the sequence of operations.

If we denote the actual cost of the ith operation by c_i and the amortized cost of the ith operation by $\hat{c}_i$, we require

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

for all sequences of *n* operations.

(3) Potential method:

The potential method is similar to the accounting method. However, instead of representing prepaid work as credit stored with specific objects in the data structure, the ***potential method*** of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations.
We associate the potential with the data structure as a whole rather than with specific objects within the data structure.

The potential method works as follows:
It starts with an initial data structure, D_0. Then n operations are performed, turning the initial data structure into D_1, D_2, D_3, ..., D_n .

Let c_i be the cost associated with the ith operation, and let D_i be the data structure that results after applying the ith operation on data structure D_{i-1}.

A **potential function** Φ maps the data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with that data structure D_i.

The **amortized cost** $\hat{c}_i$ of the ith operation is defined by:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \ .$$

The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation.

Now, the total amortized cost of the n operations is:

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \ .\end{aligned}$$

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

Thus the total amortized cost gives an upper bound on the total actual cost.

In practice, we do not always know how many operations might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i , then we guarantee that we pay in advance. We usually just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq 0$ for all i .

Over the course of the sequence of operations, the ith operation will have a potential difference of $\Phi(D_i) - \Phi(D_{i-1})$. If this value is positive, then the amortized cost $\hat{c}_i$ is an overcharge for this operation, and the potential energy of the data structure will increase. If it is negative, the amortized cost is an undercharge, and the potential energy of the data structure will decrease to pay for the actual cost of the operation.

The amortized costs defined by above equations depend on the choice of the potential function Φ. Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs.

Example-1: *(Optional)*
Consider the CollapsingFind operation on disjoint sets.

```
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
        while ( i ≠ r ) do
        {
                S := P[ i ] ;
                P[ i ] := r;
                i := S;
        }
        return r;
}
```

Suppose we perform CollapsingFind(*n*) for *n* times on a disjoint set with *n* elements represented using a tree having *n* nodes.

A cursory analysis yields a bound that is correct but not tight. A single execution of CollapsingFind takes time O(logn) in the worst case. Thus, we may wrongly conclude that a sequence of *n* CollapsingFind operations on a disjoint set with *n* elements takes time O(nlogn) in the worst case.

We can tighten our analysis to yield a worst-case cost time of O(n) for a sequence of *n* CollapsingFind operations by observing that only first CollapsingFind operation takes O(logn) time and remaining operations take only O(1) time each.

Now, the total time on the sequence of *n* CollapsingFind operations is:
O(logn)+(n-1)*O(1) = O(logn)+O(n) = O(n).

The worst-case time for a sequence of *n* CollapsingFind operations on a disjoint set with *n* elements is therefore O(n).
The average cost of each operation, and therefore the amortized cost per operation, is O(n)/n = O(1).
The amortized time = O(1) < O(logn).
Amortized Cost < Cost of Expensive Operation.

Example-2: ***Incrementing a binary up-counter*** *(Optional)*

Consider the problem of implementing a k-bit binary counter that counts upward from 0 up to 2^k-1 and back to 0, and so on. We use an array A[0 : k-1] of k bits as the counter, where A.*length* = k.
A binary number x that is stored in the counter has its lowest-order bit in A[0] and its highest-order bit in A[k-1], so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$.

Initially, x=0, and thus A[i]=0 for i=0, 1, …, k-1.

To add 1(modulo 2^k) to the value in the counter, we use the following procedure.

```
Algorithm INCREMENT(A)
{
    i=0;
    while i<k and A[i]=1 do
    {
        A[i]:=0;
        i:= i+1;
    }
    if i<k then  // that means if A[i]=0
        A[i]:=1;
}
```

Aggregate Analysis:
The cost of each INCREMENT operation is linear in the number of bits flipped (or, complemented).
A cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes O(k) time in the worst case when array A contains all 1s. Thus, we may wrongly conclude that a sequence of *n* INCREMENT operations on an initially zero counter takes O(nk) time in the worst case.

We can tighten our analysis to yield a worst-case cost of O(n) for a sequence of *n* INCREMENT operations by observing that all bits don't flip each time INCREMENT is called.

Example:
4-bit counter

Present Value of Counter A[3:0]	Incrementing Counter value by 1	Actual Cost of INCREMENT operation on Counter (number of bits flipped)
0000	0000+1 = 0001	1
0001	0001+1 = 0010	2
0010	0010+1 = 0011	1
0011	0011+1 = 0100	3
0100	0100+1 = 0101	1
0101	0101+1 = 0110	2
0110	0110+1 = 0111	1
0111	0111+1 = 1000	4
1000	1000+1 = 1001	1
1001	1001+1 = 1010	2
1010	1010+1 = 1011	1
1011	1011+1 = 1100	3
1100	1100+1 = 1101	1
1101	1101+1 = 1110	2
1110	1110+1 = 1111	1
1111	1111+1 = 0000	4
0000	0000+1 = 0001	1
Cycle Repeats		

As the above figure shows, A[0] flips each time INCREMENT is called. A[1] flips only every other time. So, a sequence of *n* INCREMENT operations on an initially zero counter causes A[1] to flip $\lfloor n/2 \rfloor$ times. Similarly, bit A[2] flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of *n* INCREMENT operations.
In general, for i=0, 1,…, k-1, bit A[i] flips $\left\lfloor \frac{n}{2^i} \right\rfloor$ times in a sequence of *n* INCREMENT operations on an initially zero counter.
The total number of flips in the sequence is thus

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$= 2n$ (according to the equation $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$.)

Therefore, the worst-case time for a sequence of *n* INCREMENT operations on an initially zero counter is O(n). The average cost of each operation, and therefore the amortized cost per operation, is O(n)/n = O(1).

Accounting Method:
Let us analyze the INCREMENT operation on a binary counter that starts at zero. As we observed earlier, the running time of this operation is proportional to the number of bits flipped, which we shall use as our cost for this example. Let us use a dollar bill to represent each unit of cost (the flipping of a bit in this example).

Note:
We assume that setting a bit to 1 costs 1\$ and resetting a bit to 0 also costs 1\$.
In the INCREMENT operation, at most one bit is set to 1.

For the amortized analysis, let us charge an amortized cost of 2\$ for each INCREMENT operation for the purpose of setting a bit to 1.
When a bit is set, we use 1\$ (out of the 2\$ charged) to pay for the actual setting of the bit, and we place the remaining 1\$ on the bit as credit to be used later when that bit is reset to 0.

At any point of time, every ‘1’ in the counter has 1\$ of credit on it, and thus we can charge nothing to reset a bit to 0; we just pay for the reset with the 1\$ bill on the bit.

The number of 1’s in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times

.

Present Value of Counter A[3:0]	Incrementing Counter value	Amortized Cost charged to INCREMENT operation	Total Credit Left	Comment
0000	0000+1 = 0001	2	2-1=1	One set
0001	0001+1 = 0010	2	(1-1)+(2-1) =1	One Reset and One Set
0010	0010+1 = 0011	2	1+(2-1) = 2	One Set
0011	0011+1 = 0100	2	(2-2)+ (2-1) = 1	Two Resets and one Set
0100	0100+1 = 0101	2	1+(2-1) = 2	One Set
0101	0101+1 = 0110	2	(2-1)+ (2-1) = 2	One Reset and One Set
0110	0110+1 = 0111	2	2+(2-1) = 3	One Set
0111	0111+1 = 1000	2	(3-3)+ (2-1) = 1	Three Resets and one Set
1000	1000+1 = 1001	2	1+(2-1) = 2	One Set
1001	1001+1 = 1010	2	(2-1)+ (2-1) = 2	One Reset and One Set
1010	1010+1 = 1011	2	2+(2-1) = 3	One Set
1011	1011+1 = 1100	2	(3-2)+(2-1) = 2	Two Resets and one Set
1100	1100+1 = 1101	2	2+(2-1) = 3	One Set
1101	1101+1 = 1110	2	(3-1)+(2-1) = 3	One Reset and One Set
1110	1110+1 = 1111	2	3+(2-1) = 4	One Set
1111	1111+1 = 0000	2	(4-4)+2 = 2	Four Resets
0000	0000+1 = 0001	2	Cycle Repeats	

Thus, for *n* INCREMENT operations, the total amortized cost is 2n. And we know that for *n* INCREMENT operations the total actual cost is 2n.

So $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ holds true. Thus, the total amortized cost bounds the total actual cost.

<u>How we have chosen amortized cost?</u>
The cost of resetting the bits within the while loop of INCREMENT procedure is paid for by the dollars on the bits that are set. The INCREMENT procedure sets at most one bit (whose cost is 1$), and therefore the amortized cost of an INCREMENT operation is at most 2$ (out of which one dollar can be used for setting a bit and second dollar can be used to reset that bit later).

<u>Potential Method:</u>
Let us again look at incrementing a binary counter. This time, we define the potential of the counter after the ith INCREMENT operation to be b_i which is equal to the number of 1's in the counter after the ith operation.
So, here $\Phi(Di) = b_i$.

Let us compute the amortized cost of an INCREMENT operation:
Suppose that the ith INCREMENT operation resets t_i bits. The actual cost (c_i) of the operation is therefore at most (t_i +1) since in addition to resetting t_i bits, it sets at most one bit to 1.
If $b_i=0$, that means the ith operation has reset all k bits, and so $b_{i-1} = t_i= k$.
If $b_i > 0$, then $b_i=b_{i-1} - t_i +1$.
In either case, $b_i \leq b_{i-1} - t_i +1$.
The potential difference is $\Phi(D_i)-\Phi(D_{i-1}) = b_i - b_{i-1}$.
$\Phi(D_i)-\Phi(D_{i-1}) = b_i - b_{i-1} \leq 1 - t_i$.

The amortized cost is therefore
$\hat{c}_i= c_i + \Phi(D_i)-\Phi(D_{i-1})$
$= (t_i +1) + \Phi(D_i)-\Phi(D_{i-1})$
$\leq (t_i +1)+(1 - t_i)$
$= 2.$

<u>Note:</u> $\Phi(D_0) = 0$.

For 4-bit counter, D_0=0000, D_1=0001, D_2=0010 and so on.

Present Value of Counter A[3:0]	Incrementing Counter value	Actual Cost (c_i) of INCREMENT operation on Counter	i	$\Phi(Di)$	$\Phi(D_i)-\Phi(D_{i-1})$	Amortized Cost $(\widehat{c_i})$ $\hat{c}_i=c_i+\Phi(D_i)-\Phi(D_{i-1})$
0000	0000+1 = 0001	1	1	1	1-0=1	1+1=2
0001	0001+1 = 0010	2	2	1	1-1=0	2+0=2
0010	0010+1 = 0011	1	3	2	2-1=1	1+1=2
0011	0011+1 = 0100	3	4	1	1-2=-1	3+(-1)=2
0100	0100+1 = 0101	1	5	2	2-1=1	1+1=2
0101	0101+1 = 0110	2	6	2	2-2=0	2+0=2
0110	0110+1 = 0111	1	7	3	3-2=1	1+1=2
0111	0111+1 = 1000	4	8	1	1-3=-2	4+(-2)=2
1000	1000+1 = 1001	1	9	2	2-1=1	1+1=2
1001	1001+1 = 1010	2	10	2	2-2=0	2+0=2
1010	1010+1 = 1011	1	11	3	3-2=1	1+1=2
1011	1011+1 = 1100	3	12	2	2-3=-1	3+(-1)=2
1100	1100+1 = 1101	1	13	3	3-2=1	1+1=2
1101	1101+1 = 1110	2	14	3	3-3=0	2+0=2
1110	1110+1 = 1111	1	15	4	4-3=1	1+1=2
1111	1111+1 = 0000	4	16	0	0-4=-4	4+(-4)=0
0000	0000+1 = 0001	1	17	1	1-0=1	1+1=2
Cycle Repeats						

Probabilistic analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm.

In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs.

Then we analyze our algorithm, computing an average-case running time, where we take the average over the distribution of the possible inputs. Thus we are, in effect, averaging the running time over all possible inputs. When reporting such a running time, we will refer to it as the ***average-case running time***.

We must be very careful in deciding on the distribution of inputs. For some problems, we may reasonably assume something about the set of all possible inputs, and then we can use probabilistic analysis as a technique for designing an efficient algorithm and as a means for gaining insight into a problem. For other problems, we cannot describe a reasonable input distribution, and in these cases we cannot use probabilistic analysis.

Example: *The hiring problem*

Consider the problem of hiring an office assistant. We interview candidates on a rolling basis, and at any given point we want to hire the best candidate we've seen so far. If a better candidate comes along, we immediately fire the old one and hire the new one.

```
HIRE-ASSISTANT(n)
1  best = 0          // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          hire candidate i
```

In this model, there is a cost c_i associated with interviewing a candidate, but a much larger cost c_h associated with hiring a candidate. The cost of the algorithm is $O(c_i n + c_h m)$, where n is the total number of applicants, and m is the number of times we hire a new person.

What is the worst-case cost of this algorithm?
Answer: In the worst case scenario, the candidates come in order of increasing quality, and we hire every person that we interview. Then the hiring cost is $O(c_h n)$, and the total cost is $O((c_i + c_h)n)$.

So far, we have mainly focused on the worst case cost of algorithms. Worst case analysis is very important, but sometimes the typical case (average case) is a lot better than the worst case.

Average case analysis of HireAssistant:

Indicator random variables:
An indicator random variable is a variable that indicates whether an event is happening. If A is an event, then the indicator random variable I_A is defined as:

$$I_A = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$$

Example: Suppose we are flipping a coin *n* times. We let X_i be the indicator random variable associated with the coin coming up heads on the ith coin flip. So

$$X_i = \begin{cases} 1 & \text{if } i\text{th coin is heads} \\ 0 & \text{if } i\text{th coin is tails} \end{cases}$$

By summing the values of X_i, we can get the total number of heads across the n coin flips.
To find the expected number of heads, we first note that

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

Now the computation reduces to computing $E[X_i]$ for a single coin flip, which is

$$\begin{aligned} E[X_i] &= 1 \cdot P(X_i = 1) + 0 \cdot P(X_i = 0) \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 \end{aligned}$$

So the expected number of heads is $\sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} (1/2) = n/2.$

Example: Let A be an event and I_A be the indicator random variable for that event. Then
$E[I_A] = P(A)$.

Analysis of HireAssistant:
We are interested in the number of times we hire a new candidate.
Let X_i be the indicator random variable that indicates whether candidate i is hired, i.e. let $X_i = 1$ if candidate i is hired, and 0 otherwise.
Let $X = \sum_{i=1}^{n} X_i$ be the number of times we hire a new candidate.
We want to find E[X], which is just $\sum_{i=1}^{n} E[X_i]$.

And we know that $E[X_i]$ = P(candidate i is hired), so we just need to find this probability. To do this, we assume that the candidates are interviewed in a random order.

Candidate i is hired when candidate i is better than all of the candidates 1 through i - 1. Now consider only the first i candidates, which must appear in a random order. Any one of them is equally likely to be the best-qualified thus far. So the probability that candidate *i* is better than candidates 1 through i - 1 is just 1/i.

Therefore $E[X_i] = 1/i$, and E[X] = = ln $\sum_{i=1}^{n} 1/i$ n+O(1).

So the expected number of candidates hired is O(ln n), and the expected hiring cost is O(c_h ln n).