UNIT-1

PART-1

Introduction

T. Maanasa

→ **Algorithm** : An Algorithm is a finite set of instructions (or) steps that accomplishes a particular task.

- **Characteristics of an Algorithm** : All the Algorithms must satisfy the following criteria.

1. **Input** : Zero (or) More quantities are externally supplied.
2. **Output** : At least one quantity is produced.
3. **Definiteness** : Each instruction is clear and Unambiguous.

Ex: Statements such as, "add 6 or 7 to X" (or) "Compute 5/0" are not permitted. (Ambiguous, Not clear)

4. **finiteness** : If we trace out the instructions of an Alg., then for all cases, the algorithm terminates after a finite no. of steps.
5. **Effectiveness** : Every instruction must be very basic & easy. Each instruction should be definite and must be feasible.

→ **process for Design & Analysis of an Algorithm** :

Creating an Algorithm is an art which may never be fully automated. There are 6 important steps in designing & Analyzing an Algorithm. It is not possible to convert an algorithm into complete working program.

1. Understand the problem
↓
2. Decide a solution
↓
3. Design an Algorithm (Iterative (or) Recursive).
↓
4. Testing (or) Validation (or) Correctness — No → back to 3
↓ yes
5. Efficiency (Time & Space Complexity) — No → back to 2
↓ yes
6. Convert into code (or) Code Implementation.

Step-1: Once a problem is given to you, first understand and Analyze that problem. When problem is clearly understood then go to step-2.

Step-2: For a given problem there may be multiple solutions. Among all the solutions we are selecting one solution.

Ex: To calculate sum of 2 numbers a & b.
There may be so many solutions like, using functions, pointers and normal addition. Among all these we are deciding a solution.

- We can write, Iterative & Recursive Algorithms.

Step-3: Once solution is decided start the designing of an algorithm. This is very important step why because, there are so many algorithm design strategies like, Divide & Conquer, Greedy Approach, Dynamic programming, Back tracking, Branch & Bound. Among all these we need to select one strategy

Step-4: Once an algorithm is designed, it is necessary to show that this Alg computes the correct answer for all the possible legal inputs. This is called as "Alg. validation". The purpose of validation is to assure us that this algorithm will work correctly independently of the issues regarding the programming language

Once the validity has been shown, a program can be written and a second phase begins. Next checking the correctness of a prg. Algorithm correctness means checking are there any errors (or) not. If more no. of errors are there again construct an Algorithm.

In correctness there are 2 ways to test an Alg.

1. Debugging - check whether the Alg contains any Errors (or) not. If any errors are there correct those errors by using some sample data set

2. Profilling — In profilling we are using the correct program & applying sample data set on that for checking the efficiency. So, profilling is a performance measurment.

Step-5: If Algorithm does not contains any Errors then we need to calculate the efficiency (or) performance which is called as "Analysis of an Alg". Analysis of an Alg. refers (or) performance Analysis refers to the task of determining how much computing time & storage an Alg requires. this is a challenging area which sometimes required great mathematical skills. we need to calculate both Time & Space complexities of an Alg.

Step-6: If both complexities are satisfied (or) lesser terms then that Alg is converted into our required programming language source code.

→ Pseudocode Notations for Expressing Algorithms:

In Computational theory, we distinguish between an Algorithm and a program. we can describe an algorithm in many ways. We can use natural language like English but resulting instructions should be definite &

Algorithm should be small & simple. Now we will present most of our Algorithms using a pseudocode that resembles C and pascal.

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces { and }.
   i. A compound statement can be represented as a block.
   ii. The body of a procedure also forms a block.
   Statements are delimited by ;
3. • An identifier (or) variable begins with a letter.
   • The data types of variables are not explicitly declared.
   • We can use simple data types such as int, float, char, boolean...
   • We can use compound data types also like Records.

```
node = record
       {
         datatype_1  data_1;
         .
         .
         datatype_n  data_n;
         node *link;
       }
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period. If 'p' points to a record of type node, "p→data_1" stands for the value of the first field in the record, "p.data_1" denotes first field of the record.

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

   Ex: i := 1;

5. There are two boolean values "true & false". In order to produce these values the logical operators are, "and, or, and not". The relational operators "$<, \leq, =, \neq, >, \geq$" are provided.

6. Arrays are defined & elements of an array are accessed using [and]. If A is a two dimensional array, the (i, j)th element of an array is denoted as A[i, j]. Array indeces start at zero.

7. The following are the available looping statements : for, while & repeat - until. • The "while" loop takes following form :

```
while <condition> do
{
    <statement 1>
    :
    <statement n>
}
```

As long as <condition> is true, the statements get executed. When <condition> becomes false, the loop is exited. The value of <condition> is evaluated at the top of the loop.

• The general form of "for" loop is :

```
for variable := value1 to value2 step step do
                                  ↳ keywords
{
    <statement 1>
    :
    <statement n>
}
```

Here "value1, value2 and step" are arithmetic value1. A "variable" of type integer (or) real value. The term "step step" is optional & taken as +1 if it doesnot occur. "step" could either be +ve (or) -ve. It will increments (or) decrement the variable by some value (or) by default as +1 (or) -1.

- A "repeat - until" statement is 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 statements execution.

- break : Results in the exit of the innermost loop.
- return : Results in the exit of the function itself & display the intermediate results of the functions.

8. A conditional statement has the following form:

if <condition> then <statement> — Simple if

if <condition> then <statement 1> else <statement 2>
└ if - else

We also employ the following "case" statement:

```
case
{
    : <condition 1> : <statement 1>
        ;
    : <condition n> : <statement n>
```

else : <statement n+1>
}

If <condition 1> is true, <statement 1> gets executed & the case statement is exited. If <condition 1> is false, <condition 2> is evaluated & case statement is exited. If none of the conditions are true, <statement n+1> is executed. This else case is optional like default

9. Input & output are done using the instructions "read & write". these will specify the input & outputs.

10. there is only one type of procedure : "Algorithm". An Algorithm consists of a heading & body. Heading as follows:

```
Algorithm Name (<parameter list>) — Header part
{
    <body part>
}
```

where "Name" is the name of the procedure & (<parameter list>) is a listing of procedure parameters. It should starts with a capital letter. the body part enclosed in { }.

Ex: An Algorithm to find & returns the maximum no of n given numbers :

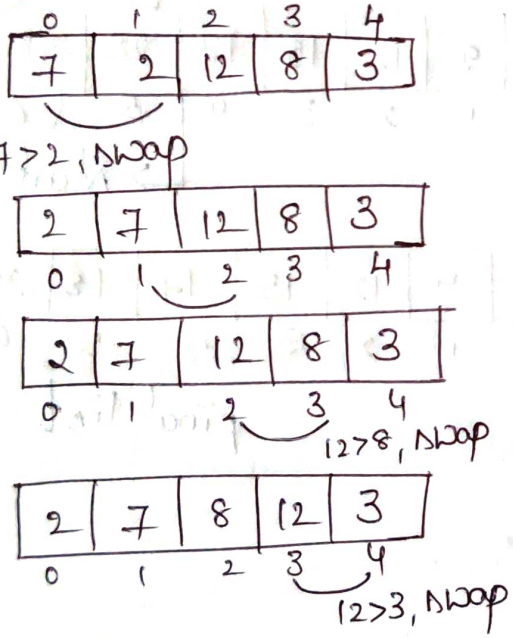
```
1. Algorithm Max(a,n)
2. // A is an array of size n. (or) a is an array of size n.
3. {
4.     Result := a[1];
5.     for i:=2 to n do
```

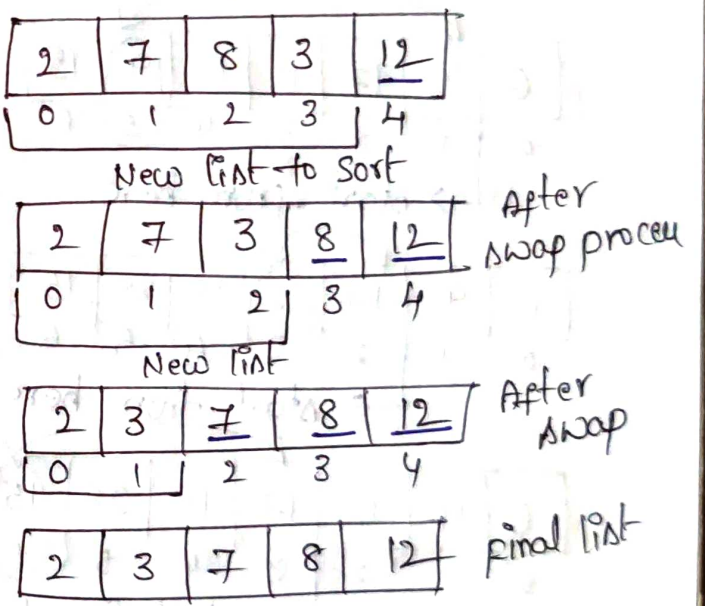
```
6.      if a[i] > Result then Result := a[i];
7.  return Result;
8. }
```

In this Alg, a & n are procedure parameters, Result and i are local variables.

Ex: Write an Algorithm to sort an array of n integers using Bubble Sort.

- the main goal of bubble sort is always trying to place largest elements at the end of an array one by one per that, in each iteration swap elements in sequence from left side.

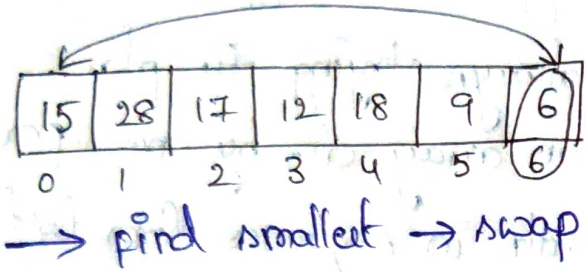


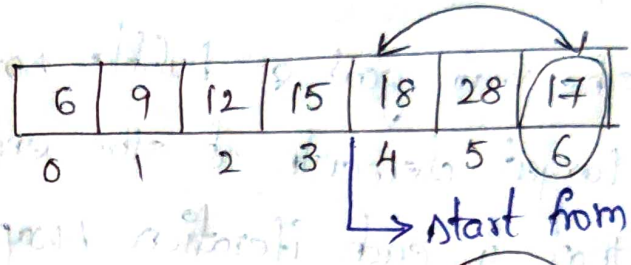


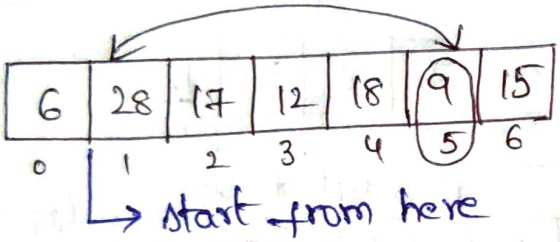

```
1. Algorithm Bubble (a,n)
2. // a is an array of size 'n'
3. {
4.     for i:=1 to n-1 do
5.     {
6.         for j:=1 to n-i do
7.         {
8.             if (a[j] > a[j+1]) then
9.                 t:= a[j]; a[j]:= a[j+1]; a[j+1]:=t;
10.        }
11.    }
12. }
```

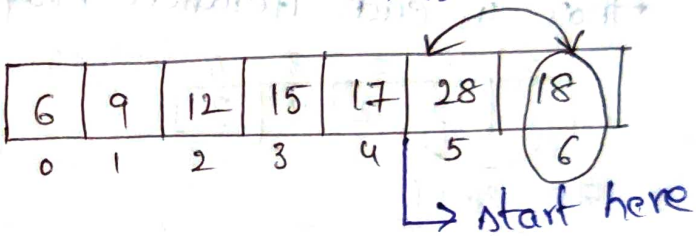
Ex: Write an Algorithm to sort an array of n integers using Selection Sort

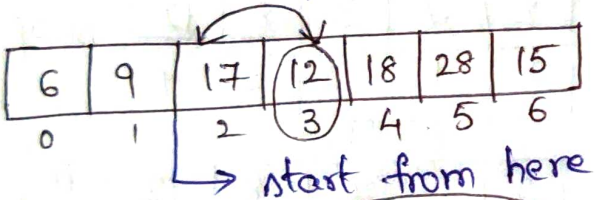
- The main goal of Selection sort is, in each iteration find one smallest integer and place that in starting positions of an array one by one by swapping. But, here elements are too far so, while swapping take care about index values.

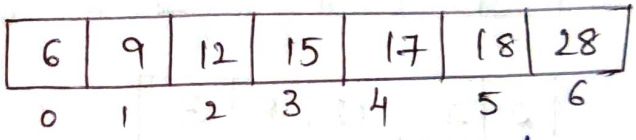




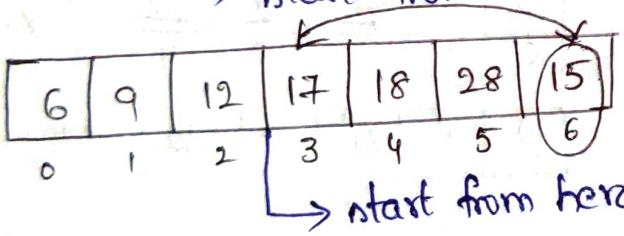








final list




```
1. Algorithm SelectionSort(a,n)
2. // Sort the array a[1:n] into ascending order.
3. {
4.     for i:=1 to n do
5.     {   j:=i;
6.         for k:=i+1 to n do
7.             if (a[k]<a[j]) then j:=k;
8.
```

```
9.      t := a[i] ; a[i] := a[j] ; a[j] := t;
10.  }
11. }
```

→ Recursive Algorithm : An Algorithm is said to be recursive, if it calls the same algorithm in its body part itself. There are 2 types of Recursive Algorithms.

- Direct Recursive : An Algorithm A is said to be direct recursive if it calls itself in it's body part.
- Indirect Recursive : An Algorithm A is said to be an indirect recursive, if it calls another algorithm B which in turn calls A.
- Every Recursive Algorithm have two elements :

i) Base Case :

- This basic statement solves the problem.
- Every recursive function must have a base case.

ii) General Case :

- Recursive function of an Algorithm.
- Each function call reduces the size of the problem.

— Designing a Recursive Algorithm :

- Determine the base case.

- Determine the general case.
- finally combine the base & general case into an Alg

Ex: Write a Recursive Algorithm for Towers of Hanoi problem.

Base case: Move one disk from source to destination.

General case: Move n-1 disks from source to intermediate disk.

Problem Definition: three towers a, b, c are there. Source tower A is having some disks need to send all disks one by one to the destination tower C by taking help of intermediate tower B. (Auxiliary storage).

- Move (n-1) disks from source tower (A) to an intermediate tower (B), using 'C'.
- Move one disk from Source (A) to destination tower (C).
- Move (n-1) disks from intermediate tower (B) to the destination tower (C), using 'A'.

— Algorithm TowersOfHanoi (n, A, B, C)

```
// Move 'n' no. of disks from tower 'A' to tower 'C'
{
  if (n>=1)
  {
    TowersOfHanoi (n-1, A, C, B);
    Write ("Move top disk from tower", A, "to tower", C);
    TowersOfHanoi (n-1, B, A, C);
  }
}
```

Ex: Write a Recursive Alg for factorial of a no. calculation.

Base Case: Factorial (0) = 1.

General Case: factorial (n) = n * Factorial (n-1)

— Algorithm Factorial (n)

```
{ if (n==0) return 1;
  else return n * Factorial (n-1);
}
```

Ex: Write a Recursive Alg to find the Fiboracci Series.

Base Case: Fib (0) = 0 & Fib (1) = 1.

General Case: Fib (n) = Fib (n-1) + Fib (n-2).

— Algorithm Fib (n)

```
{ if (n==0 or n==1) return n;
  else return Fib(n-1) + Fib (n-2) ;
}
```

→ Performance Analysis: Performance evaluation can be loosely divided into 2 major phases:

Priori Analysis	Posterior Testing
1. After writing an Alg. measuring the performance of an Alg. before programming	1. After completion of code implementation testing of a program.
2. Language Independent: No any programming language	2. Language Dependent: Based upon programming language

to write an Algorithm.

3. Hardware Independent: No need to consider any H/W configuration while writing an Alg.

4. Performace is measured in Time & Space functions.

source code size will vary.

3. Hardware Dependent: Execution speed & Memory space required based upon the H/W configuration.

4. Performance is measured in Exact Time & Space values.

- Prior Analysis: To Analyze the performance of an Algorithm 2 measures are there: Space & Time Complexity

→ Space Complexity: Space Complexity of an Algorithm is the amount of memory (RAM) it needs to run to completion. We can divide RAM space into 2 parts:

1. Fixed Space: Consists of space needed by the instructions, Constants and Variables. It is denoted by 'c'.

2. Variable Space: consists of space needed by compound variables (Arrays) whose size is dependent on the particular problem. It is denoted by Sp.

∴ Space complexity, $S(P) = c + Sp$, where c is constant

- We can write 2 types of Algs: Iterative & Recursive. for both the types we will see Performance Analysis.

Ex: Algorithm abc(a,b,c)

```
{
    return a+b*c+(a+b-c)/(a+b)+4.0;
}
```

Space Complexity

a → 1 unit
b → 1 unit
c → 1 unit

$S(P) \geq 3$ units

- In this simple Alg., a,b,c are the constants. only fixed space is there occupied by a,b,c. Each constant will occupy 1 unit (or) cell (or) byte (or) word of fixed space. '≥' symbol denotes small amount of space for instructions also.

Ex: Iterative Alg. To calculate sum of 'n' elements of an Array

```
1. Algorithm Sum(a,n)
2. {
3.     s:=0.0;
4.     for i:= 1 to n do
5.         s:= s+a[i];
6.     return s;
7. }
```

Space Complexity

fixed space: n → 1 unit, s → 1 unit, i → 1 unit

a → n units (variable space).

- In this algorithm fixed space occupied by n, s, i & variable space occupied by an Array. Because, initially we don't know what is the Exact size of an array. It will vary based on the problem.

$S_{sum}(n)$ = Fixed + Variable space

= 3 + n ∴

$\therefore S_{sum}(n) \geq n+3$ units.

Ex: ~~Iterative Alg~~

Recursive Alg to calculate sum of n elements of an Array.

```
1. Algorithm RSum(a,n)
2. {
3.     if (n≤0) then return 0.0;
4.     else return RSum(a,n-1) + a[n];
5. }
```

- Space Complexity : In Recursive Alg. first we need to calculate space required by Recursive function call.

(i.e) RSum(a,n) → 1 (a[n]) + 1 (n) + 1 (return address) = 3 Units.

RSum(a,n-1) → 1 (a[n-1]) + 1 (n) + 1 (return address) = 3 Units.

⋮

RSum(a,n-n) → 1 (a[n-n]) + 1 (n) + 1 (return add.) = 3 Units.

- To calculate the space complexity of total Algorithm there are 2 Methods : Tree & Stack Methods.

— Tree Method ⇒ for n=3,

RSum(a,3)
calls ↓
RSum(a,2)
calls ↓
RSum(a,1)
calls ↓
RSum(a,0) → condition fails here.

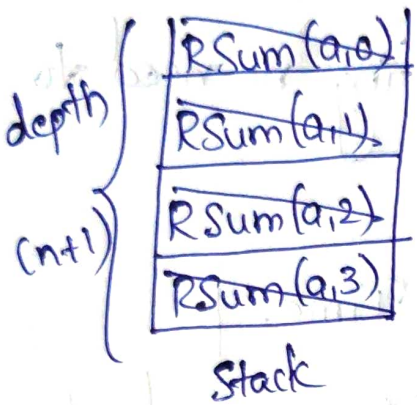
With n=3, 4 Recursive Calls.
(n+1)
Height (or) Depth of tree : (n+1)

$\therefore S_{RSum}(n) \geq 3(n+1)$ Units.

3 → Space Complexity of each function call

(n+1) → total Recursive calls

— Stack Method ⇒ for n=3,



→ Condition fails

∴ push each function call into the stack, whenever condition fails calculate the function, pass that value to the next function call using return address.

After calculating each function, pop that function call from the stack, finally make the stack as empty & returns the final result

$\therefore S_{RSum}(n) \geq 3(n+1)$ Units.

3 → Each function call; (n+1) → Depth of the stack.

Ex: Algorithm Ex(a,n)
{ int i;
for i:=1 to n do
A[i]:=0;
}

Size of an Array 'n' - 1 Unit.
Array size of 'n' elements - n Units.
loop variable 'i' - 1 Units

$S(Ex) \geq n+2$ units

Ex: Algorithm Ex(a,b,n)
{
for i:=1 to m do

Size of m,n - 2 Units
Array Size (a,b) - mn Units.

```
    for i:=1 to n do
        b[i,j] := a[i,j];
}
```

loop Variables i, j – 2 Units

∴ S(Ex) ≥ mn+4 Units

→ Time Complexity : Time Complexity T(P) is taken by an Algorithm (or) Program P is the sum of the Compile time & Run time. While calculating no need to consider Compile time.

T(P) = Compile Time + Execution Time.

- there are 2 Methods in calculating Time Complexity :

1) Step Count Method : In this method we need to count what are the simple statements & Executable statements. Assign '0' unit of Execution time to simple statements & '1' unit of time to Executable statements. Finally take the sum of Execution time.
   - for Algorithm Heading → 0 units
   - for Brace{ → 0 units
   - for Expressions → 1 unit
   - for conditional Statement → 1 unit
   - for loop statement → no. of times loop is repeating

Ex: Algorithm abc (a,b,c) → 0
{ → 0

```
return a+b*c+(a+b-c)/(a+b)+4.0;   --> 1
}                                  --> 0
                                   ------
                                   1 Unit
```

Ex:
```
Algorithm Sum(a,n)                 --> 0
{                                  --> 0
  s:=0.0;                          --> 1
  for i:=1 to n do                 --> (n+1)
  s:=s+a[i];                       --> n
  return s;                        --> 1
}                                  --> 0
                                   ------
                                   2n+3 units
```

- In for loop (n+1) units Execution Time means, 'n' denotes execution time required for all true cases (i.e). i=1 to n, 'i' is the execution time for one false case.
- How many no. of times a for loop is executing, those many times statements & innermost for loops will also execute

Ex:
```
Algorithm Sum(a,n)
{
  s:=0;
      count:=count+1; // for assignment statement
  for i:=1 to n do
  {   count:=count+1; // for for
      s:=s+a[i];
  }   count:=count+1; // for assignment statement.
```

```
    count := count + 1 ;  // for last time of for
    return s ;
    count := count + 1 ;  // for return.
}
```

2n+3 steps

Note: Step count tells us how the run time for a prg changes with changes in the instance characteristics.

Ex: Addition of two m×n matrices.

```
Algorithm Add (a, b, c, m, n)
{
  for i := 1 to m do ———————— (m+1) units
  {
    for j := 1 to n do ———————— m(n+1) units
    {                            (outer for loop, inner loop)
      c[i, j] := a[i, j] + b[i, j] ; ———— mn units
    }                              (outer for loop, inner for loop)
  }
}
```

Total Time Complexity = m+1 + mn+1 + mn

$\therefore t(p) = 2mn + 2m + 1$ steps (or) units.

— Space Complexity : A - mn (for 'i' - m, 'j' - n units)
B - mn (" ")
C - mn (" ")
m - 1 unit

n – 1 Unit for cell
i – 1 Unit
j – 1 Unit

$\therefore S(p) \geq 3mn + 4$ Units.

Ex: Algorithm Multiply (A, B, C)

```
{
  for i := 0 to m do ———————— (m+1) Units
  {
    for j := 0 to n do ———————— m(n+1) Units
    {
      c[i,j] := 0; ———————— mn Units
      for k := 0 to n do ———————— mn(n+1) Units
      {
        c[i,j] := c[i,j] + A[i,k] * B[k,j]; — m×n×n
      }
    }
  }
}
```

Total Time Complexity = $m+1+mn+m+mn+mn^2+mn+mn^2$

$\therefore T(p) = 2mn^2 + 2mn + 1$

– Space Complexity:

A – mn cells
B – mn cells
C – mn cells
m – 1 cell
n – 1 cell

i – 1 cell
j – 1 cell
k – 1 cell

$\geq 3mn + 5$ cells

Ex: Recursive Alg. Time Complexity calculation Using Step-count Method.

```
Algorithm RSum (a,n)
{
    if (n≤0) then return 0.0;
    else return RSum (a,n-1) + a[n];
}
```

- We obtain the following recursive formula for 'RSum'.

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

- Using Substitution Method need to solve this formula.

$$\therefore t_{RSum}(n) = 2n+2 \text{ steps.}$$

-
```
Algorithm RSum (a,n)    // Using 'count' variable
{
    count := count +1 ;  // for 'if' condition
    if (n≤0) then
    {
        return 0;
        count := count +1 ; // for the 'return'.
    }
    else
    {
        return RSum (a,n-1) + a[n];
        count := count+1 ;  // for addition, function
    }                       // call & return.
}
```

2) Frequency Count Method : Tabular Method.

Ex: Iterative sum of 'n' numbers.

Statement	s/e	Frequency	Total Steps
1. Algorithm Sum(a,n)	0	—	0
2. {	0	—	0
3. S:=0;	1	1	1
4. for i:=1 to n do	1	n+1	n+1
5. S:=S+a[i];	1	n	n
6. return S;	1	1	1
7. }	0	—	0
Total			2n+3

Ex: Recursive Sum of 'n' numbers.

Statement	s/e	Frequency		Total Steps	
		n=0	n>0	n=0	n>0
1. Algorithm RSum(a,n)	0	—	—	0	0
2. {	0	—	—	0	0
3. if (n≤0) then	1	1	1	1	1
4. return 0;	1	1	0	1	0
5. else return	0	0	0	0	0
6. RSum(a,n-1)+a[n];	1+x	0	1	0	1+x
7. }	0	—	—	0	0
Total				2	2+x

Time Complexity $T(n) = 2$ & $2+x$ where, $x = t_{RSum}(n-1)$.

Recurrence Relation. $T(n) = \begin{cases} 2 & \text{if } n=0 \\ 2 + t_{RSum}(n-1), & \text{if } n>0 \end{cases}$

$\therefore T(n) = 2 + T(n-1)$

$= 2 + 2 + T(n-2)$ (Using Substitution Method, $T(n-1) = 2 + T(n-2)$)

$= 2 + 2 + 2 + T(n-3)$.

$\vdots$

$= 2 + 2 + 2 + T(n-n) = 2n + T(0) = \underline{2n+2}$.

Ex: Algorithm for Matrix ~~Multiplication~~ Addition.

	Statement	s/e	Frequency	Total steps
1	Algorithm Add (a,b,c,m,n)	0	—	0
2	{	0	—	0
3	for i:=1 to m do	1	m+1	m+1
4	for j:=1 to n do	1	m(n+1)	mn+m
5	c[i,j]:=a[i,j]+b[i,j];	1	mn	mn
6	}	0	—	0
	Total			2mn+2m+1

In this frequency count Method, table need to be constructed with 4 columns. First column is an Alg. statements, second column is whether statement is a general (or) an executable statement, third column is how many no. of times a statement is executing, last column is multiplication of 2nd & 3rd columns.

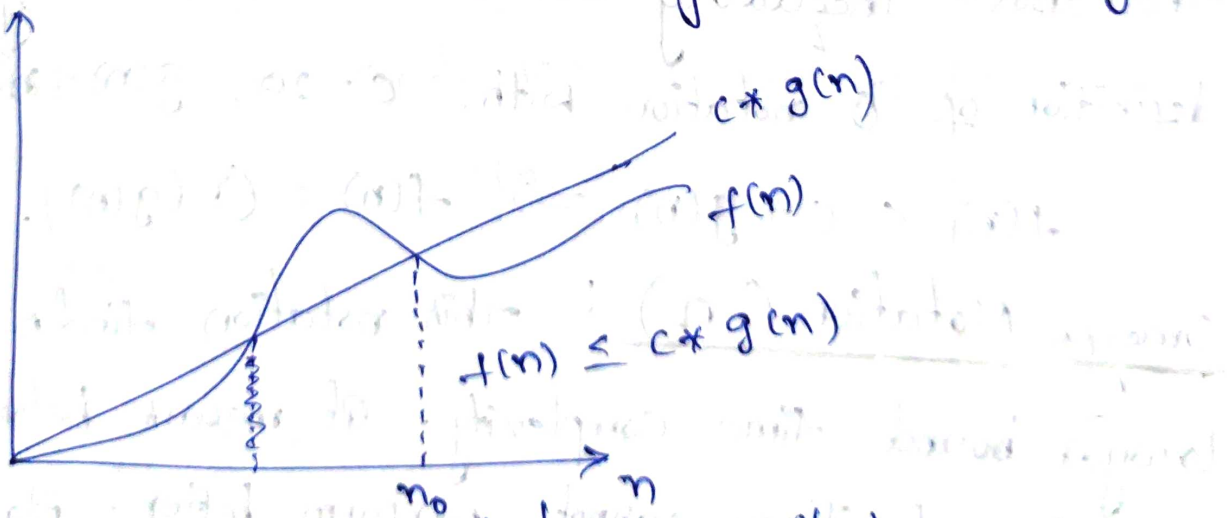
→ **Asymptotic Notation**: Asymptotic means study of function of parameter 'n' & 'n' becomes larger & larger without bound. Here we are concerned with how the running time of an alg. increases with the size of i/p.

Asymptotic notations is an other method of finding time complexity for an algorithm. Notations are,

- **Big-Oh Notation (O)**: This notation (O) gives an Upper bound & a function $f(n)$. This notation represents worst case time complexity which consume more computing & Running time (i.e). $g(n)$. Above $f(n)$ time alg. cannot perform better.

$f(n) = O(g(n))$ such that, 2 +ve constants 'c' & 'n_0' with condition that : $f(n) \leq c * g(n)$ $\forall$ n, n_0, c.

consumes less time complexity ↓ consumes more time complexity



Ex: Given $f(n) = 3n+2$, prove that $f(n) = O(g(n))$.

Sol: $f(n) = 3n+2$. Assume that, $g(n) = n$, $c = 5$.

$3n+2 \leq 5n \quad \forall n \geq 1$.

$3(1)+2 \leq 5(1) \Rightarrow 5 \leq 5$.

$3(2)+2 \leq 5(2) \Rightarrow 8 \leq 10$.

⋮ ⋮

The above inequality can be satisfied using the definition of 'O' notation with, $c=5$, $g(n)=n$ & $n_0=1$

$$f(n) \leq c * g(n) \Rightarrow f(n) = O(g(n)).$$

Ex: Given $f(n) = 20n^3 - 3$, prove that $f(n) = O(n^3)$.

Sol: $f(n) = 20n^3 - 3$. Assume that, $g(n) = n^3$, $c = 20$.

$20n^3 - 3 \leq 20n^3 \quad \forall n \geq 1$.

$20(1)^3 - 3 \leq 20(1)^3 \Rightarrow 17 \leq 20$.

$20(2)^3 - 3 \leq 20(2)^3 \Rightarrow 157 \leq 160$.

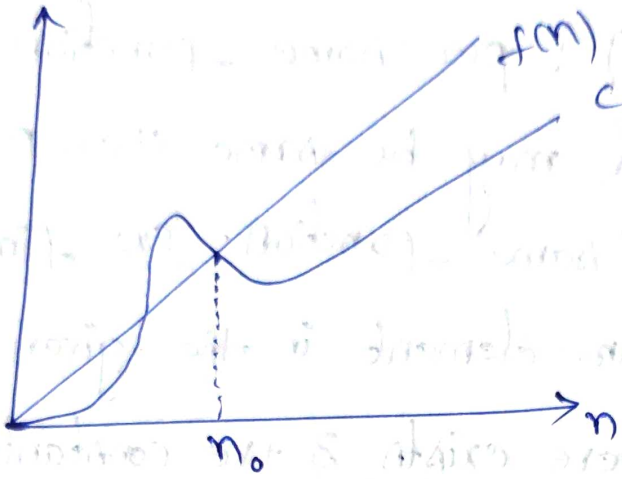
⋮ ⋮

The above inequality can be satisfied using the definition of 'O' notation with, $c = 20$, $g(n) = n^3$, $n_0 = 1$.

$$f(n) \leq c * g(n) \Rightarrow f(n) = O(g(n)).$$

- **Omega Notation (Ω)**: This notation finds the lower bound time complexity. It means below this time the algorithm cannot perform better. Algorithms with least time complexity is represented with Ω.

$f(n) = \Omega(g(n))$ if there exists 2 +ve integer constants 'c' & n_0 with condition, $\boxed{f(n) \geq c * g(n)}$ $\forall$ $n, n \geq n_0$.



n_0 - particular i/p.

n - total i/ps.

Ex: Given $f(n) = 3n+2$ prove that $f(n) = \Omega(g(n))$.

Sol: $3n+2 \geq 3n$ $\forall$ $n \geq 1$, Assume, $g(n) = n$, $c = 3$

$3(1)+2 \geq 3(1) \Rightarrow 5 \geq 3$

$3(2)+2 \geq 3(2) \Rightarrow 8 \geq 6$

⋮

The above inequality is satisfied according to Ω definition by setting, $c = 3$ & $n_0 = 1$.

$f(n) \geq c * g(n) \Rightarrow f(n) = \Omega(g(n))$.

Ex: Given $f(n) = 50n^2 + 10n - 5$ prove that $f(n) = \Omega(g(n))$

Sol: $50n^2 + 10n - 5 = f(n)$, Assume, $g(n) = n^2$, $c = 50$.

$50n^2 + 10n - 5 \geq 50n^2$ $\forall$ $n \geq 1$.

$50(1)^2 + 10(1) - 5 \geq 50(1)^2 \Rightarrow 55 \geq 50$

$50(2)^2 + 10(2) - 5 \geq 50(2)^2 \Rightarrow 215 \geq 200$

⋮

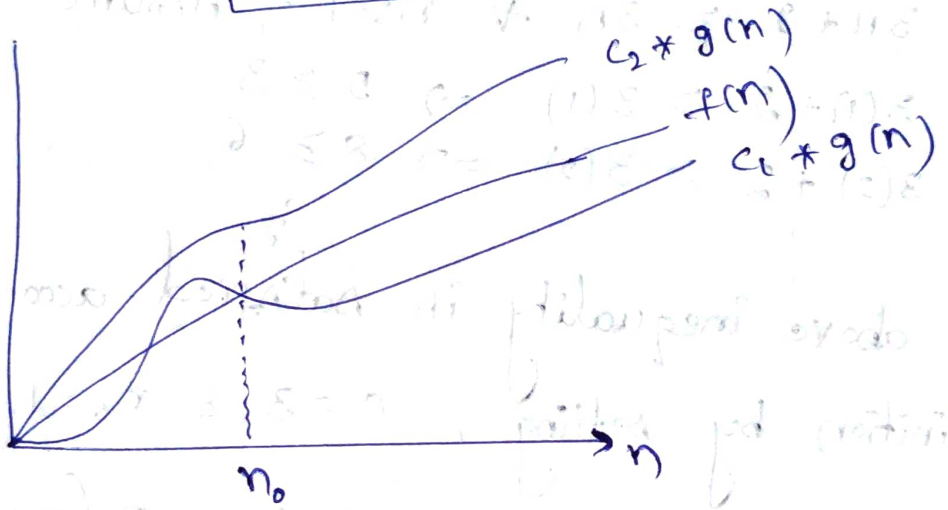
The above inequality is satisfied according to Ω

definition by setting with $c = 50$ & $n_0 = 1$.

$f(n) \geq c * g(n) \Rightarrow f(n) = \Omega(g(n))$.

- **Theta Notation (Θ)** : For some function the lower & upper bound may be same (i.e). Big-oh & omega will have the same function. Ex: find the maximum (or) minimum element in the given array.

$f(n) = \Theta(g(n))$ if there exists 3 +ve constants c_1, c_2 & n_0 with condition, $\boxed{c_1 * g(n) \leq f(n) \leq c_2 * g(n)}$ $\forall\ n \geq n_0$.



Ex: prove that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Sol: To prove above statement determine +ve constants c_1, c_2, n_0. Such that, $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ $\forall\ n, n_0$.

Divide by n^2 throughout, $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$.

When $n \geq 1$ & $c_2 \geq \frac{1}{2}$, inequality $\frac{1}{2} - \frac{3}{n} \leq c_2$, hold good.

When $n \geq 7$ & $c_1 \leq \frac{1}{14}$, then $c_1 \leq \frac{1}{2} - \frac{3}{n}$, hold good.

$\therefore c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ & $n_0 = 7$

$f(n) = \Theta(g(n))$.

- **Little-Oh Notation (o)** : Besides O, Ω & Θ, there are two other greek symbols o & ω. We use o-notation to denote upper bound that is not asymptotically tight.

$f(n) = o(g(n))$ (read as $f(n)$ is equal to little oh of $g(n)$) if $\boxed{f(n) < c * g(n)}$ for any +ve constant $c > 0$, $n_0 > 0$ & $n \geq n_0$

Ex: $f(n) = 2n$ prove that $f(n) = o(g(n))$.

Sol: $f(n) = 2n$. Assume $g(n) = n^2$, $c = 1$.

$2n < n^2$ $\forall$ $n \geq n_0$.

$2(1) < 1^2 \Rightarrow 2 < 1$ ✗, $n_0 = 2 \Rightarrow 4 < 4$ ✗

$2(3) < 3^2 \Rightarrow 6 < 9$ ✓

The above inequality is satisfied using definition 'o' with, $c = 1$, $g(n) = n^2$. at $n \geq 3$.

$f(n) < c * g(n) \Rightarrow f(n) = o(g(n))$.

— Another definition for o-notation. $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

- **Little Omega Notation (ω)** : This notation denotes a lowyer bound that is not asymptotically tight.

$f(n) = \omega(g(n)) \Rightarrow \boxed{f(n) > c * g(n)}$ for $c > 0$, $n_0 > 0$ & $n \geq n_0$.

— Another definition for ω-notation : $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$.

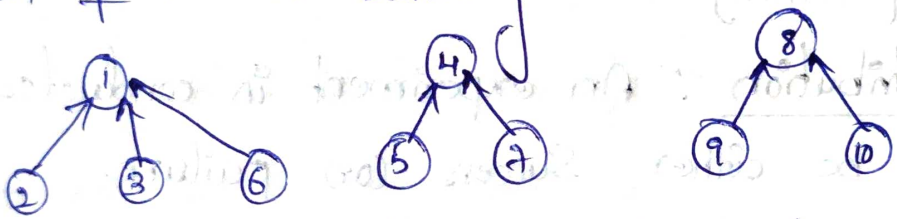
→ **Disjoint Sets** : A disjoint set is a data structure that contains partitioned sets. Two sets A & B are disjoints if they have no elements in common. $A \cap B = \phi$.

A collection $S = \{S_i\}$ of non empty sets forms a partition of a set 'S' if the sets are disjoint (i.e) $S_i \cap S_j = \phi$ & their union is S (i.e). $S = \bigcup_{i=1}^{n} S_i$

Ex: Consider a set $S = \{1, 2, 3, 4 \dots 10\}$. These elements can be partitioned into 3 disjoint sets.

$A = \{1, 2, 3, 6\}$, $B = \{4, 5, 7\}$, $C = \{8, 9, 10\}$.

Each of set is denoting as a trees.



Array representation of above trees is:

i	1	2	3	4	5	6	7	8	9	10
parent	-1	1	1	-1	4	1	4	-1	8	8

Root nodes don't have parent node. parent = -1.

— **Disjoint Set Operations** : Operations are as follows:

i) **MIN**: Determines minimum no. of elements in the set

$P = \{1, 2, 3, 4, 5\}$ & $Q = \{4, 5\} \Rightarrow \min\{P, Q\} = 2$.

ii) **Delete**: To delete given element from the set.

Ex: $P = \{4,5,6,7\}$ & $Q = \{10,11,12\}$ then to delete '6' first find the set that contains the element '6'.

find (6) = P, Now delete 6 from set P (i.e).

delete (6) ⇒ Resulting Set $P = \{4,5,7\}$.

iii) FIND: To find the set to which the given element 'i' belongs to.

Ex: $P = \{1,2,3\}$ & $Q = \{4,5\}$ then find (4) = Q.

iv) UNION: To combine all the elements in two sets.

Ex: $P = \{1,2,3\}$ & $Q = \{4,5,6\} \Rightarrow P \cup Q = \{1,2,3,4,5,6\}$.

v) INSERT: To determine common elements of 2 sets

Ex: $P = \{3,4,5,6\}$ & $Q = \{4,6,7,8\} \Rightarrow P \cap Q = \{4,6\}$.

→ Union & Find Algorithms:

UNION operation: UNION (i, j), it means the elements of set i & set j are combined. If we want to represent union in form of a tree: then UNION (i, j), i is the parent, j is child represented as,

Ex: UNION (1,3)

UNION (2,5)

- Algorithm Union (i, j)

```
{  integer i, j;
}  PARENT (j) ← i;
```

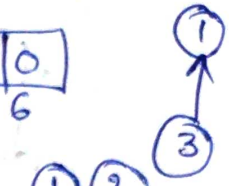
Ex: UNION (1,3), UNION (2,5), UNION (1,2)

Initially PARENT array contains '0':

1	2	3	4	5	6
0	0	0	0	0	0

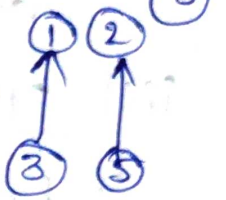
UNION (1,3) ⇒ PARENT (3) ← 1

1	2	3	4	5	6
0	0	1	0	0	0



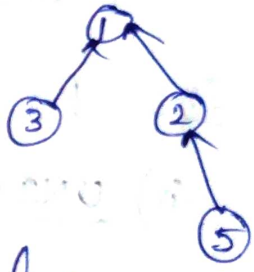
UNION (2,5) ⇒ PARENT (5) ← 2

1	2	3	4	5	6
0	1	1	0	2	0



UNION (1,2) ⇒ PARENT (2) ← 1

1	2	3	4	5	6
0	1	1	0	2	0



Time Complexity : Time taken for UNION is constant, all n-1 unions can be proceed in O(n).

FIND Operation : FIND (i) implies that it finds the root node of ith node, it finds the name of set i.

Ex: UNION (1,3) ⇒

FIND (1) = 1

FIND (3) = 1 Since its parent is '1' (i.e. root node is 1).

- Algorithm FIND (i) // find root of tree containing element

```
{
  integer i, j;
  j ← i;
  while PARENT(j) > 0 do
  j ← PARENT (j);
  repeat
  return (j);
}
```

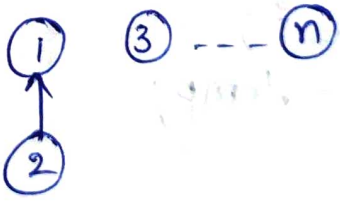
→ Weighted Rule for UNION (i,j) : Although the SimpleUnion (i,j) & SimpleFind (i) algorithms are easy to state but their performance characteristics are poor.

Ex: ① ② ③ ④ ---- ⓝ If we want to perform following sequence of operations Union (1,2), Union (2,3) ---- Union (n-1,n). the sequence of Union operations generate tree as below.

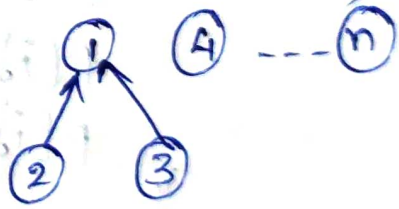
(n-1) sequence of union can be proceed in time O(n). we can improve the performance of Union & find by avoiding the creation of tree by applying weighting rule for union

① ② ③ ---- ⓝ

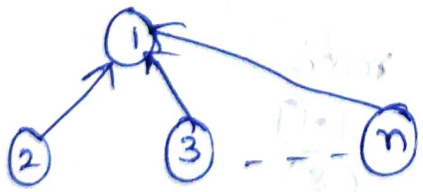
Union (1,2)



Union (1,3)



Union (1,n)



To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain

"count" field in root of every tree count[i] denotes no. of nodes in tree with root i.

Since all nodes other than roots have +ve numbers in parent (P) field, we can maintain count in P field of the root as -ve number.

Algorithm WeightedUnion (i, j)

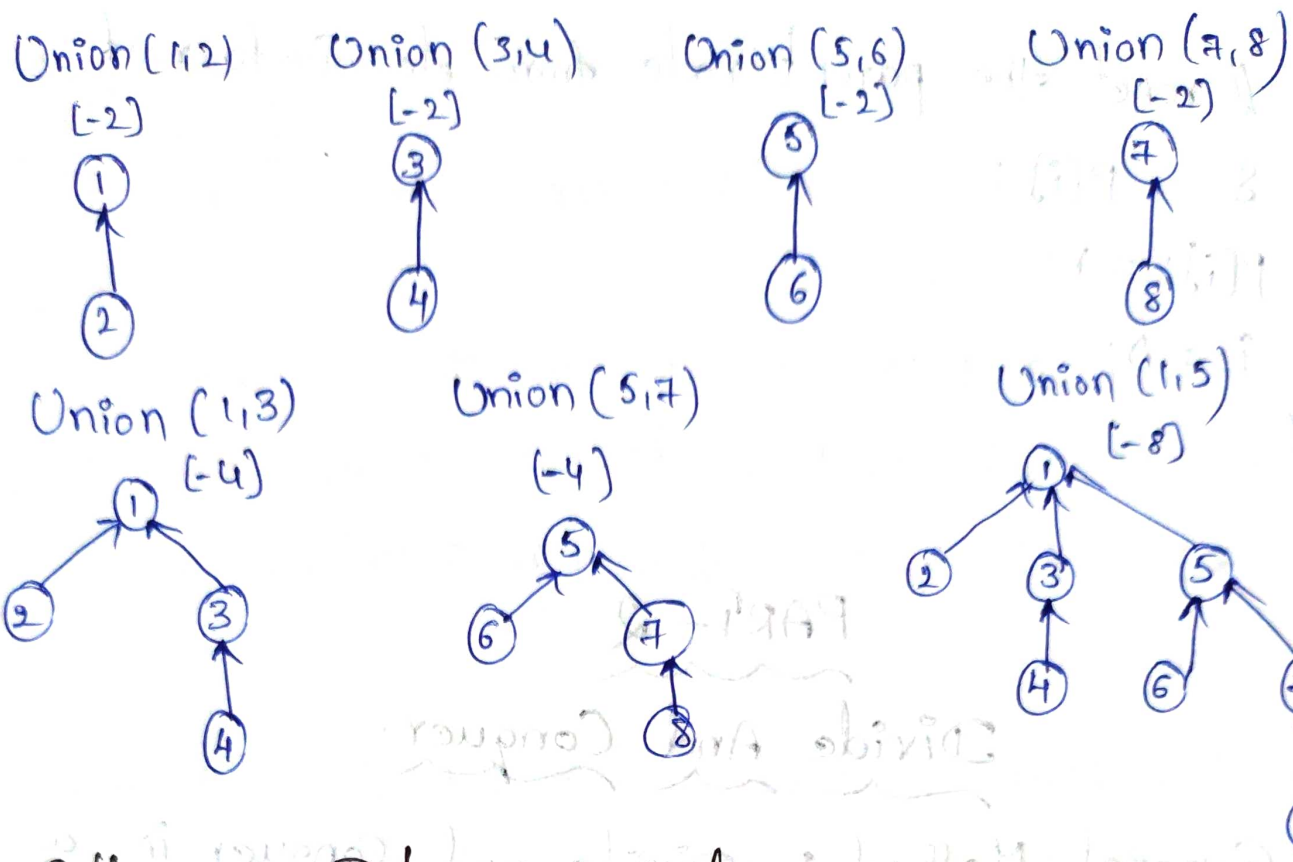
// Union sets with roots i & j, i ≠ j using weighted rule.

// P[i] = -count[i] and P[j] = -count[j].

```
{
    temp := P[i] + P[j];
    if ( P[i] > P[j] ) then
    {
        // i has fewer nodes
        P[i] := j;
        P[j] := temp;
    }
    else
    {
        // j has fewer nodes
        P[j] := i;
        P[i] := temp;
    }
}
```

Ex: Union Operations with counts at roots.

[-1]	[-1]	[-1]	[-1]	[-1]	[-1]	[-1]	[-1]
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)



→ Collapsing Rule for find : We can improve the performance of find by avoiding Extra work & Moves to find an element from tree.

Ex: When collapse find is used to perform find(8) in above tree requires going up three links & resetting three links. Directly we can connect 8 to root '1'. With in single move i can find '8' from '1'.

Algorithm CollapsingFind (i)
// find the root of the tree containing element 'i'.
// Use collapsing rule to collapse all nodes from i to root.
{
  r := i;
  while (P[r] > 0) do r := P[r]; // find root.
  while (i ≠ r)
  {

```
// reset the parent node from ele. i to root
    s := P[i];
    P[i] := r;
    i := s;
  }
}
```