

Unit 1

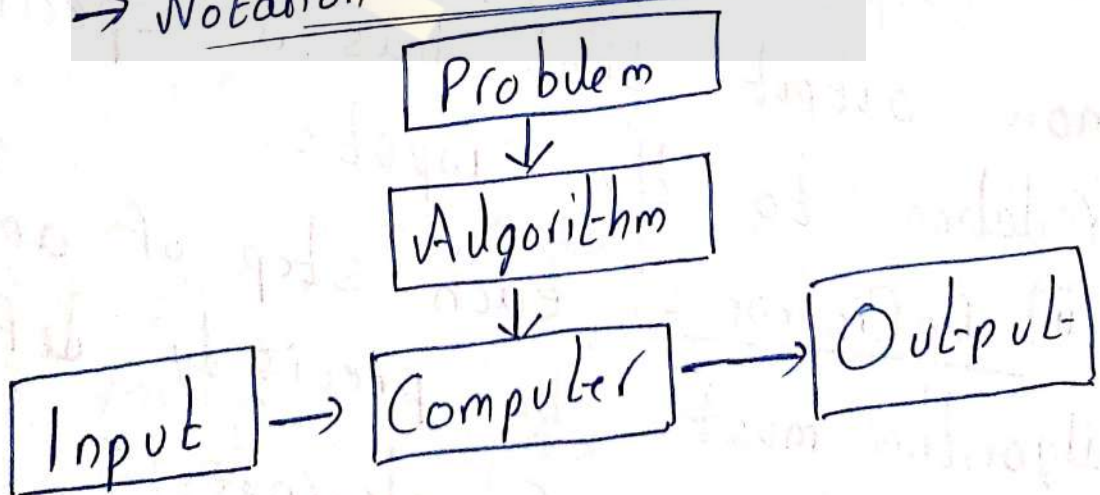
Introduction to Algorithms & elementary Data Structures

→ Algorithm:

Algorithm is a Finite sequence of well defined, computer implementable instructions typically to solve a class of problems or to perform computation.

(or)
Algorithm is typically refers to a set of instructions that can be executed by a computer to produce the desired result.

→ Notation of algorithm:



→ to solve a particular problem we can write multiple algorithms

ex to sort an array of numbers
→ heap sort, insertion sort etc...

→ The range of inputs which that algorithm will accept should be specified

ex $0 \leq T \leq 10^6$

→ Every algorithm must be unambiguous.

→ Properties of Algorithm:

i) Input :- An algorithm has zero or more inputs are given to it initially before the algorithm begins or dynamically as the algorithm runs.

ii) Output :- An algorithm has one or more outputs that has a specific relation to the inputs.

iii) Definiteness :- each step of an algorithm must be precisely defined.
(unambiguous & clearness).

iv) Finiteness :- An algorithm must always terminate after a finite number of steps, each of which may require one or more operations.

v) Effectiveness :- An algorithm is also generally expected to be effective, in the sense that its operations must all be sufficiently basic (i.e., can be solved by someone on paper using pen/pencil).

→ Algorithm development life cycle :-

→ Problem Definition :- Understand problem

→ Constraints & Conditions :- Understand constraints (conditions) if any.

→ Design Strategies (Algorithmic strategy).

→ Express & Develop the algo (on paper)

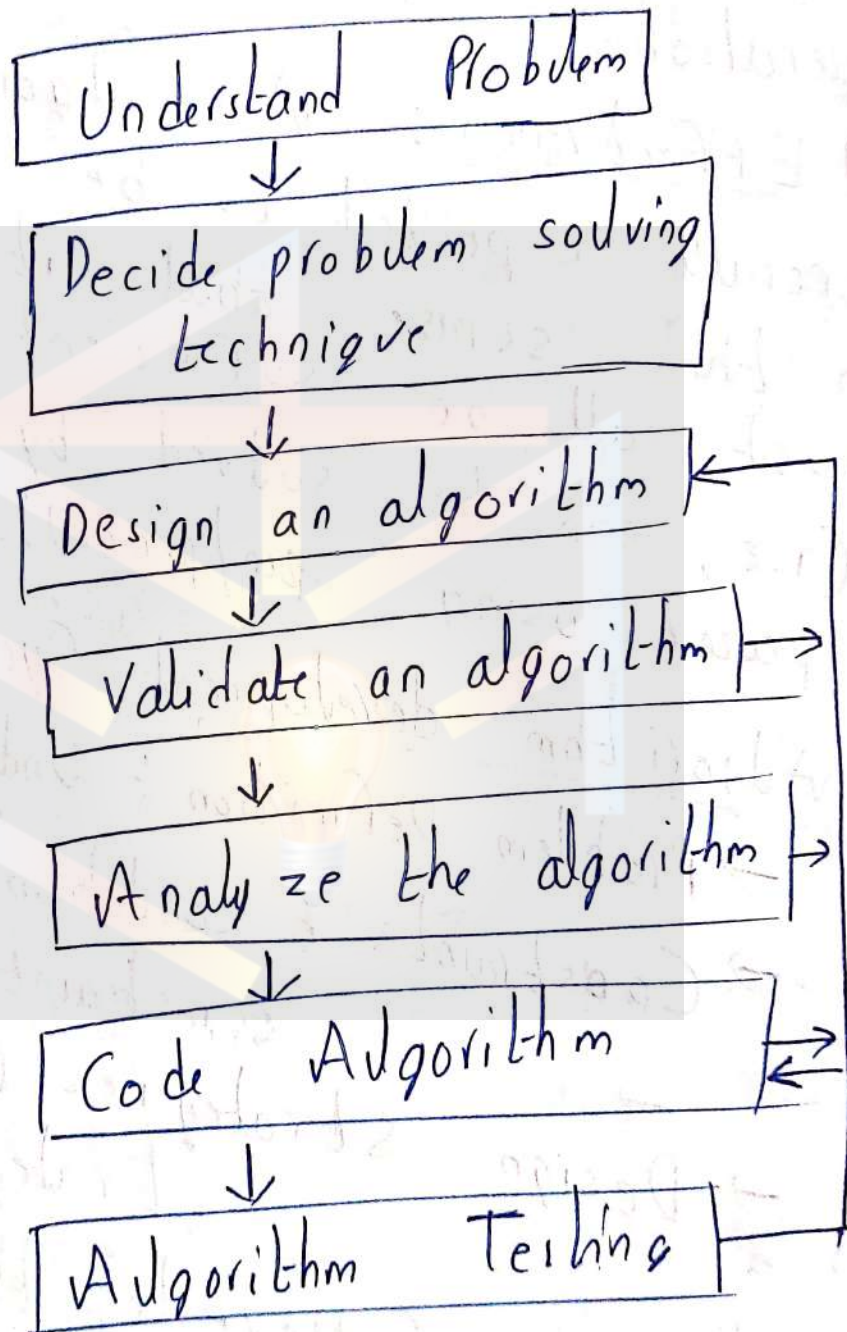
→ Validation (Dry run on paper)

* → Analysis (space & Time complexity)

→ Coding

- Testing & Debugging
- Installation
- Maintenance

Flow:



→ Pseudo code For expressing Algorithms:

↳ A hypothetical language used to define algorithms is pseudo language (pseudo code)

Note: pseudo code is not a programming language.

→ General procedure to write the pseudo code.

i) All comments must begin with // & continue until the end of line.

ii) Blocks are indicated with matching braces { and }.

A compound statement can be represented as a block - each statement end with ;

Procedure Name

```
{  
  statement - 1;  
  statement - 2;  
  ⋮  
  statement - n;  
}
```


iii) data type declaration & all the same datatype data-name;
int, float
char etc.,

even compound data types can be formed with records.

ex node = record

```
{  
    datatype-1 data-1;  
    datatype-2 data-2;  
    ...  
}
```

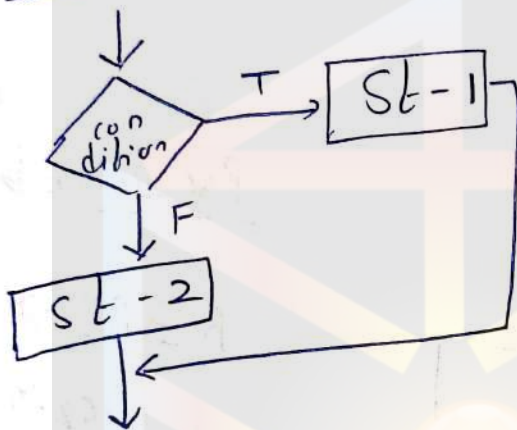
iv) Assignment of values to the variables is done using
:= or ←

v) There are two boolean values true & false. (to produce these values we use logical operators (and, or, not) and the relational operators ($<$, \leq , \neq , \geq , $>$) are provided.)

vii) elements of multidimensional arrays are accessed using [and]
 $a[1, 2]$

viii) a) selection logic: (decision logic)
if else, if else if else if then

if else Flow chart:-



Pseudo code

IF condition THEN

st 1 ;

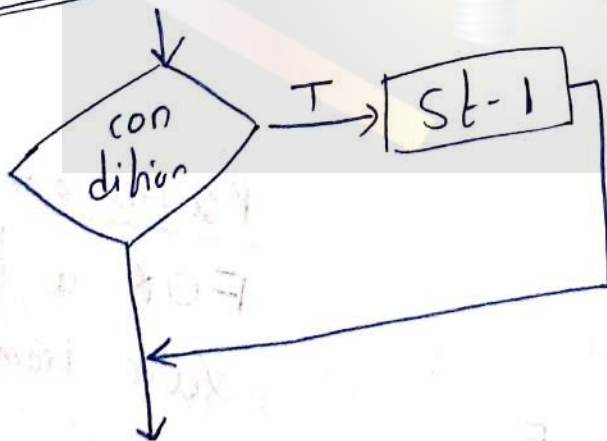
ELSE

st 2 ;

END IF

if then
if else if
else

Flow chart:-



Pseudo code:-

IF condition THEN

st 1 ;

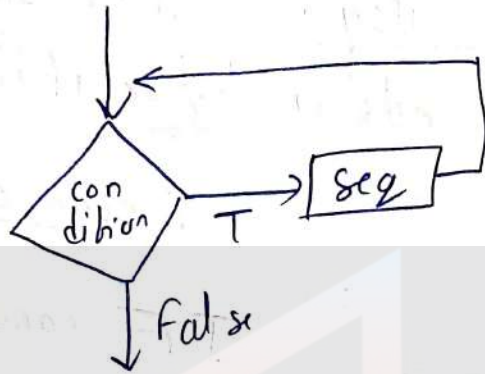
END IF

similarly we can do for
if else if else... & switch

b) Iterative logic (loops)

while

Flow chart



Pseudo code

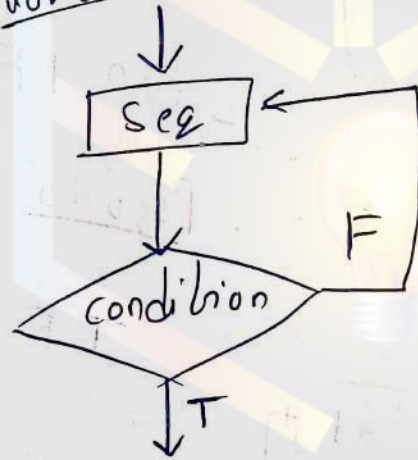
WHILE condition

Sequence

END WHILE

Repeat until

Flowchart



Pseudo code

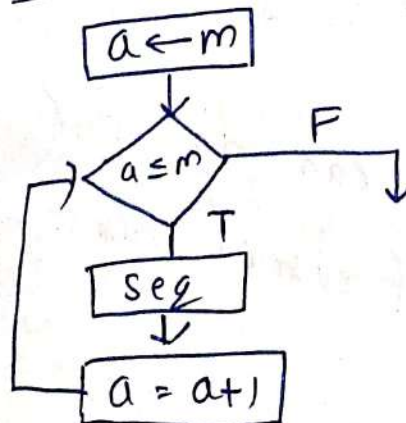
REPEAT

Sequence

UNTIL condition

For block;

Flowchart



Pseudo code

FOR a taking
value from m to n

Sequence

END FOR

viii) Here we see procedure as algorithm

Algorithm Name (parameter list)

{

Pseudo code

}

Note: rules

→ Writing only 1 statement per

line

→ Capitalize initial keywords

(IF, ELSE, END IF, WHILE,
END WHILE, REPEAT, UNTIL)

→ Indent to show hierarchy

→ Input & output should be mentioned

ex: i) Write an algorithm for finding the

maximum & minimum
of a given list of
numbers

- a) algorithm to find max
- b) algorithm to find min

Algorithm : Maximum(a)

{

n := a.length;

max := a[0];

For i := 1 to n-1 do

{ if (a[i] > max) then

max := a[i];

}

return max;

}

Algorithm : Minimum(a)

{

n := a.length

min := a[0];

For i := 1 to n-1 do

{

if (a[i] < min) then

min := a[i];

}

return min;

}

input : (a)

output : nothing
(return
max)

input : (a)

output : nothing
(return
min).

→ Analysis of Algorithm:-

we do analysis of algorithm to do a performance comparison b/n different algorithm to figure out which one is best possible option

- Time
- Space (@ M.M.)
- Bandwidth
- Register
- Battery power.

→ Types of Analysis:-

i) Experiment or Apostrium analysis

→ Analysis of algorithm

after it is converted to code.
→ Analysis b/n two algorithm can be done by running them with different inputs & see which one takes less time.

Adv:- We get exact values not rough or approx values.

disadv: Final result depends on background software & hardware, programming language, even the temp of the room.

ii) A priori or Independent analysis

→ We do analysis using asymptotic notation & mathematical tools of an algorithm i.e before converting into program.

→ Here we evaluate the performance of an algorithm in terms of input size. We calculate how does the time (or space) will be taken by an algorithm increases with the input size.

(we select an algorithm which best in long term (as increase in inputs))

→ Space Complexity:

→ The space complexity of an algorithm indicates the amount of temporary storage required for running the algorithm. i.e., it is the amount of memory needed by the algorithm to run to completion.

$$S(P) = C + SP$$

↑ Space of problem, algorithm, program ↑ Fixed part Independent of I/O ↑ variable part (dynamic). ↓ dependent on I/O

- Space of code
- Space for simple variable
- Fixed size of components
- Space for constant

- space for reference variables
- recursion stack space.

ex: i) Algorithm: $xyz(x, y, z)$

```
{  
  return (x + y + z);  
}
```

$(1 + 1 + 1) = 3$

$$S(p) = 3 + 0 = 3$$

ii) Algorithm: ADD (a, n)

{

$s := 0;$

For $i = 1$ to $n;$

$s := s + a[i];$

return $s;$

$a \rightarrow$ inside loop 1 to n
 $\Rightarrow n$

$n \rightarrow$ 1 unit

$s \rightarrow$ 1 unit

$i \rightarrow$ 1 unit

$$S(p) = C + S_p$$

3 + n units

iii) Algorithm sum (a, n)

{ if ($n \leq 0$) then
return 0;

else
return ($a[n] + \text{sum}(a, n-1)$)

depth of recursion $\Rightarrow 0$ to n
 $\Rightarrow 0, 1, 2 \dots n$
 $\Rightarrow n+1$

per each recursion.

$a \rightarrow 1 \text{ unit}$

$n \rightarrow 1 \text{ unit}$

return $\rightarrow 1 \text{ unit (as recursion)}$.

$$\Rightarrow 3 * (n + 1)$$

\rightarrow Space complexity is not so important when compared to time complexity.

\rightarrow Time complexity:

The time complexity of an algorithm is an amount of time required by algorithm to run to completion.

Compile Time $\swarrow \searrow$ Run Time.

✓
we consider only run time.

\rightarrow the time complexity is given in terms of frequency count. coz it

is not possible to compute time complexity using physical clock

→ Frequency count is basically a count denoting no. of times of execution of statements.

ex: i) Algorithm sum(a, n)

FC

{

$s := 0$

→ 1

For $i := 1$ to n

→ $n+1$ ($0 \dots n$ & $n+1$)

$s := s + a[i];$

→ n ($0 \dots n$)

return s ;

→ 1

}

$2n+3$

ii) For ($i=1$; $i < n$; $i = i * 2$)

{

~~ret~~

→ no. of
execution

}

$i = 1$
 $i = 2$
 $i = 4$
 $i = 8$
...

$2^k = n$

$\Rightarrow k = \log n$

iii) For ($i = n ; i \geq 1 ; i = i/2$)

{

→ no. of
excursion!

}

ex $i = 16$

$i = 8$

$i = 4$

$i = 2$

$i = 1$

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\Rightarrow k = \log n$$

iv) For ($i = 1 ; i * i < n ; i++$)

{

→ no. of excursion!

}

$i = 1$

$i = 2$

$i = 4$

$i = 16$

$$i^2 = n$$

$$\Rightarrow i = \sqrt{n}$$

v) $p = 0$
For ($i = 1 ; i < n ; i = i * 2$)

{

$p++$

}

For ($j = 1 ; j < p ; j = j * 2$)

{

}

$$\rightarrow p = \log n$$

$$\rightarrow \log p$$

$$\Rightarrow \log(\log n)$$

→ Worst Case Analysis:

In worst case analysis we calculate upper bound on running time of an algorithm. We must know the case that cause max no. of operation to be executed.
(max time)

ex: Quicksort takes maximum time on a sorted i/p array
 $O(n^2)$

→ Best Case Analysis:

In the best case analysis we calculate the lower bound on running time of an algorithm. We must know the case that cause minimum no. of operations to be executed.
(min time)

ex: Quicksort min time
→ $O(n \log n)$

→ Average case Analysis:

In average case analysis, we take all possible input & calculate computing time for all of the inputs. do average to get result (complex to calculate).

→ Order Notation / Asymptotic notation

→ The graph b/n lower & upper bound

→ To choose the best algorithm we need to check efficiency of algorithm.

→ efficiency is measured using time complexity.

→ Asymptotic notation is a way to represent time complexity.

→ (big O) O .

→ (Omega) Ω .

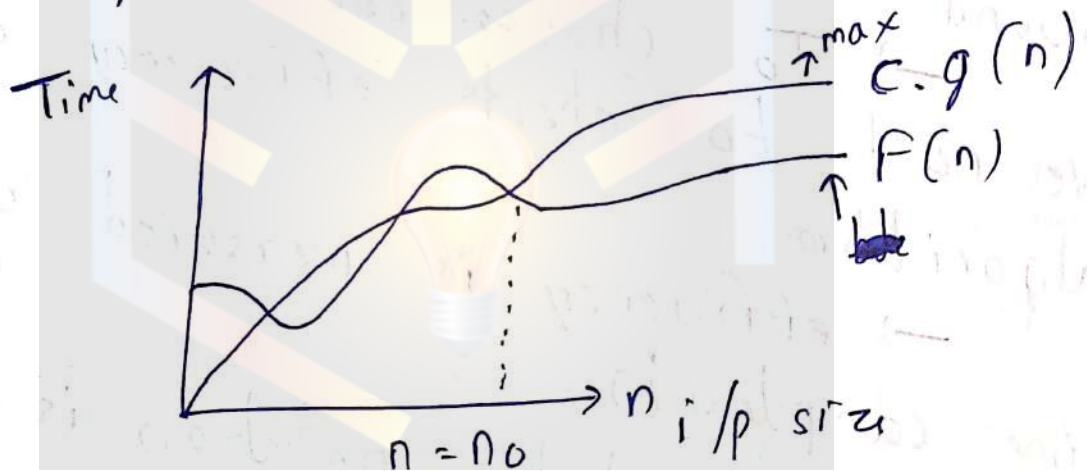
→ (Theta) Θ .

i) Big O (O)

This notation is used to represent upper bound of algorithm running time (max time) (worst case time).

The function $F(n) = O(g(n))$

if & only if there exist positive constants C & n_0 such that $F(n) \leq C \cdot g(n) \forall n$ where $n \geq n_0$.

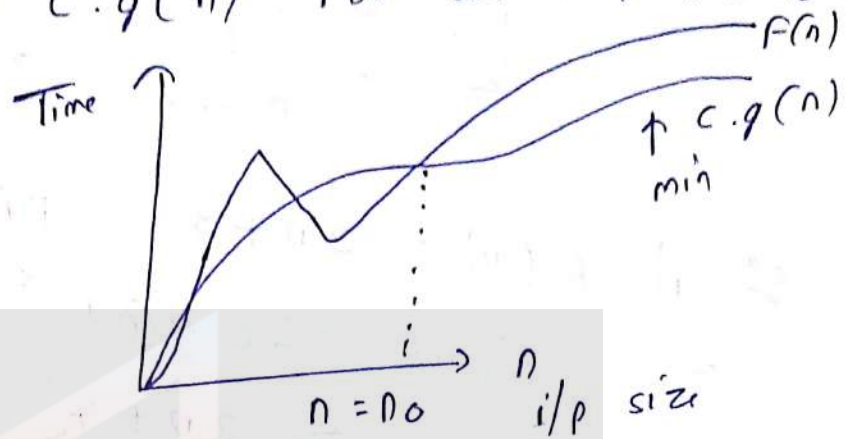


ii) Omega (Ω)

It is used to denote the lower bound of an algorithm.

(min time)
(best case)

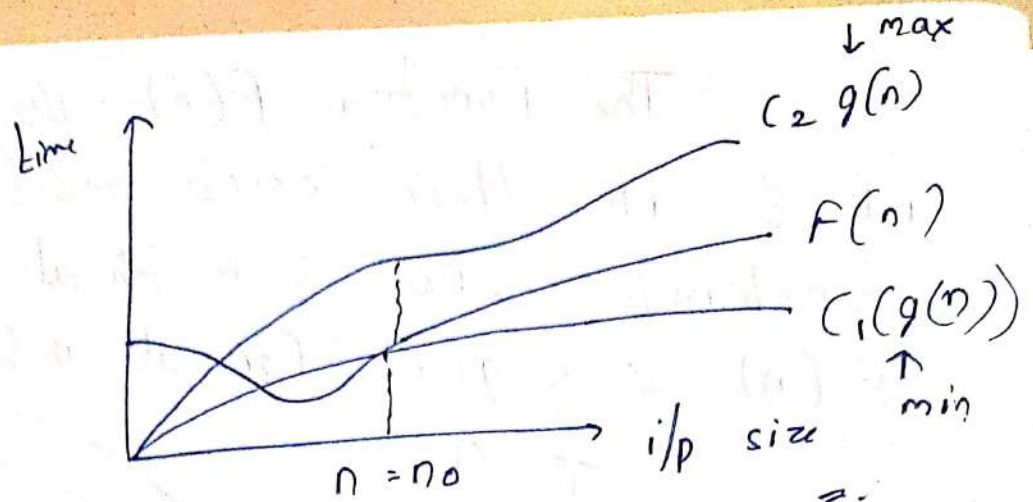
The function $F(n) = O(g(n))$ if & if there exist positive constant c , no such that $F(n) \leq c \cdot g(n)$ for all n & $n \geq n_0$



iii) Theta (Θ) :

It is used to denote the average running time of algorithm.

The function $F(n) = \Theta(g(n))$ if & if there exist positive constant c_1, c_2, n_0 such that $c_1 \cdot g(n) \leq F(n) \leq c_2 \cdot g(n)$ for all n & $n \geq n_0$.



ex: i) $F(n) = 2n + 3$ Big $O(\overline{O})$

$$\begin{aligned}
 & F(n) \leq C \cdot g(n) \\
 & 2n + 3 \leq 2n + 4 \quad \checkmark \\
 & \quad \quad \quad \text{next} \quad \quad \quad \downarrow \\
 & \quad \quad \quad = 2n + n \\
 & \quad \quad \quad = 3n \\
 & 2n + 3 \leq 3n \quad \checkmark \rightarrow \text{next} \\
 & \quad \quad \quad \downarrow \quad \downarrow \\
 & \quad \quad \quad C \quad g(n) \\
 & O(g(n)) \Rightarrow O(n)
 \end{aligned}$$

ii) $F(n) = 4n^2 + 5$ (10)

$$\begin{aligned}
 & F(n) \leq C \cdot g(n) \\
 & 4n^2 + 5 \leq 4n^2 + 6 \\
 & \quad \quad \quad \downarrow \\
 & \quad \quad \quad 5n^2 \\
 & \quad \quad \quad \downarrow \\
 & \quad \quad \quad g(n)
 \end{aligned}$$

$$O(g(n)) = O(n^2)$$

$$\text{iii) } F(n) = 2n + 3 \quad (\Omega)$$

$$F(n) \geq c \cdot g(n)$$

$$2n + 3 \geq 2n + 2$$

$$\begin{array}{c} \vdots \\ 2n \\ \swarrow \quad \searrow \\ c \quad g(n) \end{array}$$

$$\Omega(g(n)) = \Omega(n)$$

$$\text{iv) } F(n) = 4n^2 + 5 \quad (\Omega)$$

$$F(n) \geq c \cdot g(n)$$

$$4n^2 + 5 \geq 4n^2 + 4$$

$$\begin{array}{c} \vdots \\ 4n^2 \\ \swarrow \quad \searrow \\ c \quad g(n) \end{array}$$

$$\Omega(g(n)) = \Omega(n^2)$$

$$\text{v) } F(n) = 2n + 3 \quad (\Theta)$$

$$c_1 \cdot g(n) \leq F(n) \leq c_2 \cdot g(n)$$

$$2n + 2 \leq 2n + 3 \leq 2n + 4$$

$$\begin{array}{c} \vdots \qquad \qquad \vdots \qquad \qquad \vdots \\ 2n \qquad \qquad 2n+3 \qquad \qquad 3n \\ \swarrow \quad \searrow \qquad \swarrow \quad \searrow \\ c_1 \quad 2n \quad g(n) \quad c_2 \quad 3n \quad g(n) \\ \therefore \Theta(n) \end{array}$$

vi) $F(n) = 4n^2 + 5$ (O)

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$$4n^2 + 4 \leq 4n^2 + 5 \leq 4n^2 + 6$$

$$4n^2$$

\swarrow
 \searrow

c_1
 $g(n)$

$$\begin{array}{c} 5^n \\ \downarrow \quad \searrow \\ c_2 \quad g(n) \end{array}$$

$$\Rightarrow O(n^2)$$

Note:

$$O(1) = \text{constant}$$

$O(n)$ = linear

$O(n^2)$ = Quadratic

$O(\log n) = \log$ arithmetic

$O(n^3)$ = cubic

$O(2^n) \Rightarrow$ exponential.

ex true or false

(Time) $O(1) < O(\log n) < O(n) < O(n \log n)$

(True) $O(n^2) < O(2^n) < O(n^2)$

→ Small notation : (without = in all previous cases).

Every thing is same as big notations just we take strictly increasing or monotonically increasing case & equal case is not allowed.

ex) i) given $F(n) = 2n$ then prove

$$F(n) \neq \Theta(n^2)$$

<u>ex</u>	$n = 1$	n^2
	$\Rightarrow 2$	1
	2	
	$n = 100$	100×100
	$\Rightarrow 200$	
	$n = 1000$	1000×1000
	$\Rightarrow 2000$	
	\vdots	\vdots
	\downarrow	\downarrow
\therefore	$F(n) \not\leq$	$O(n^2)$
		True

(n log n)

→ Elementary Data Structure:

→ refer basics of stack, Queue theory & program (not in syllabus but refer them).

SW
OS
Unit-2

Time complexity: worst case

Stack:-

Search

$O(n)$

Queue:-

$O(n)$

Insert

$O(1)$

Deletion

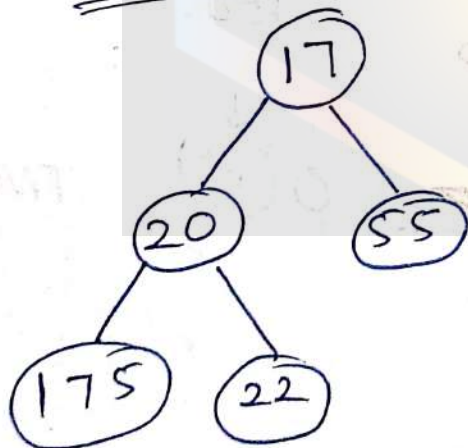
$O(1)$

$O(1)$

Heaps & Heap sort:-

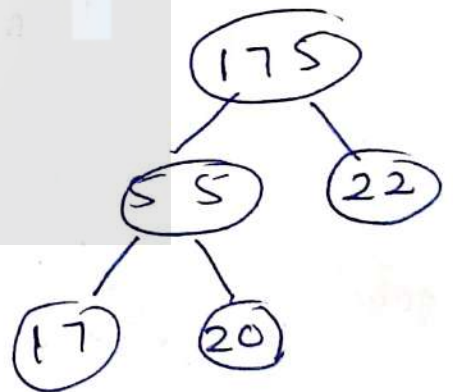
→ A complete binary tree with min or max value at top (root) (L → R)

ex: Ascending (min Heap)



min heap: each parent node is always minimum

Descending (max heap)



max heap: each node parent is always maximum

→ Deletion always occur on root node

algorithm:

heap-sort (A)^{array}

{
 build-max-heap (A) ①

build
max heap

For $i \leftarrow \text{length}[A]$ down to 2

{
 do exchange ($A[1] \leftrightarrow A[i]$)
 heap-size [A] \leftarrow heap-size [A] - 1
 max-heapify ($A, 1$) ②

Delete
max
element
& store
at end
we get
ascending
order

}

}

① build-max-heap (A)

{

① heap-size [A] \leftarrow length [A]

For $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ down to 1

② {
 do max-heapify (A, i)

}

max-heapify (A, i)

index of
left &
right child {
 $L \leftarrow \text{left}[i] // 2i$
 $R \leftarrow \text{right}[i] // 2i + 1$
if ($L \leq \text{heap-size}[A]$ &
 $A[L] > A[i]$)

check left
& root
which is
max

$\text{largest} \leftarrow L$

else

$\text{largest} \leftarrow i$

check
right- &
max.
which is
max

if ($R \leq \text{heap-size}[A]$ &
 $A[R] > A[\text{largest}]$)

$\text{largest} \leftarrow R$

if ($\text{largest} \neq i$)

swap if
needed

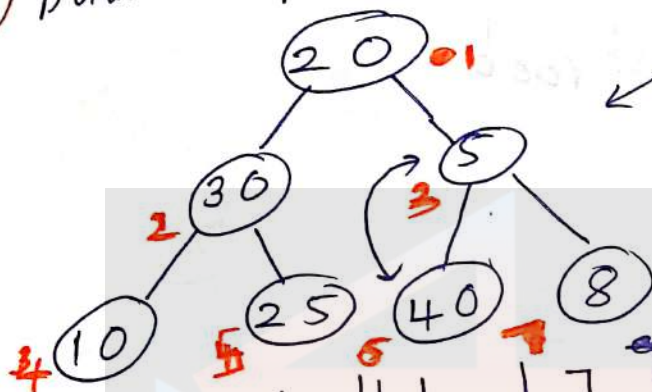
{
Exchange $[A[i] \leftrightarrow A[\text{largest}]]$
max-heapify (A, largest)
}

}

ex: max heap (20, 30, 5, 10, 25, 40, 8).
 ① → construct max heap (Build-max-heap(A))

A →	20	30	5	10	25	40	8
	1	2	3	4	5	6	7

① build complete tree

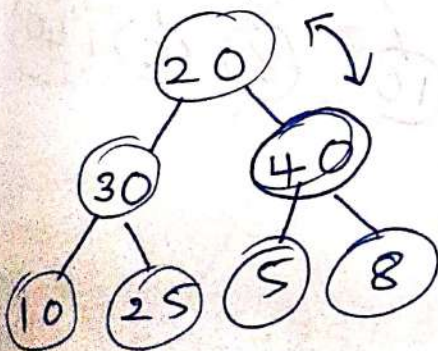


just a representative for fast understanding.

② $i = \left\lfloor \frac{\text{length}}{2} \right\rfloor = \left\lfloor \frac{7}{2} \right\rfloor = \left\lfloor 3.5 \right\rfloor = 3$
 (3, 2, 1)

@ 3

40 > 5 ✓
~~swap~~
 max = 40
 40 < 8 ✗
 ∴ max = 40
 swap.



@ 2

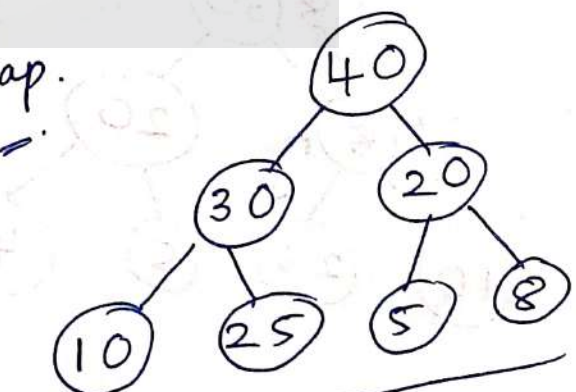
10 > 30 ✗
 max = 30
 30 < 25 ✗
 max = 30

same
 so
 no
 swap.

@ 1

30 > 20 ✗
 max = 30
 40 > 30 ✓
 max = 40

swap

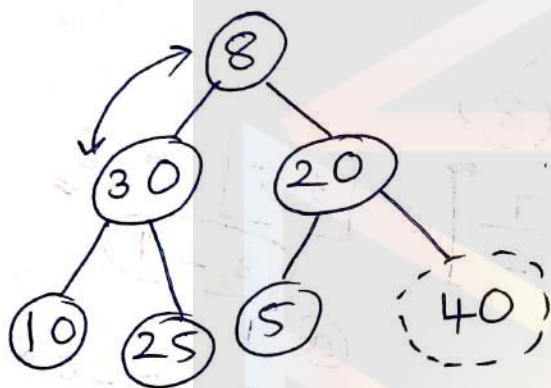


max heap of given data.

② → applying sort (heap-sort(A))
continue ②

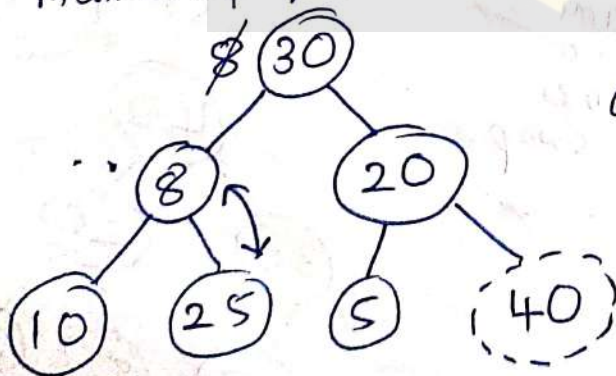
From length to 2
→ 7 to 2
7, 6, 5, 4, 3, 2, 1

@ 7
swap 7th with 1st root.

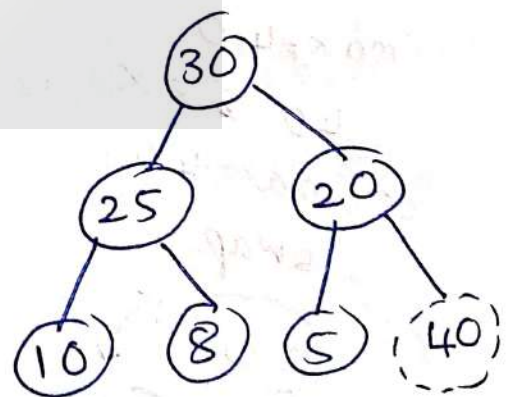


now length → 7-1
⇒ 6

→ restore heap properties
max-heapify(A, 1)



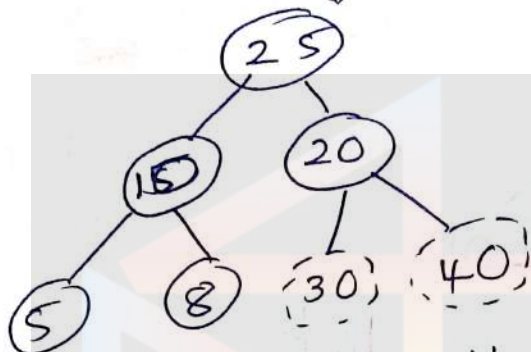
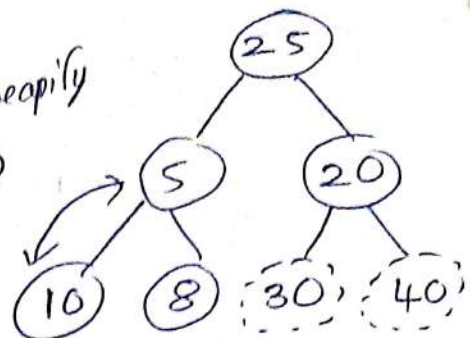
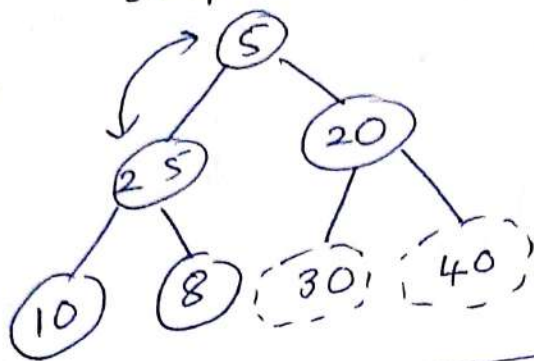
again ⇒



@ 6

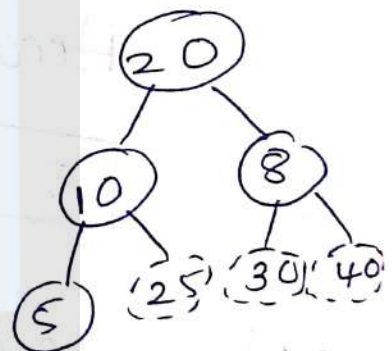
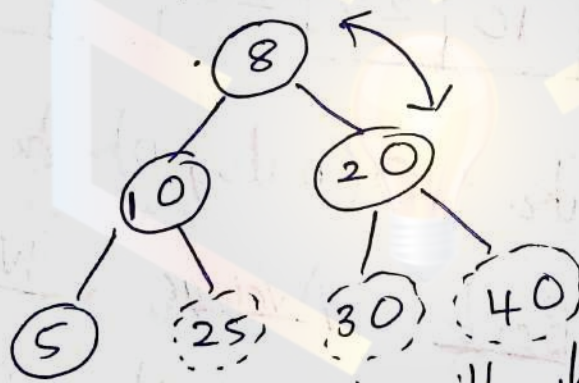
swap root with last

max-heapify



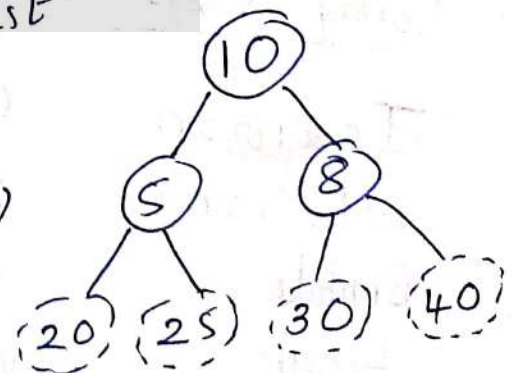
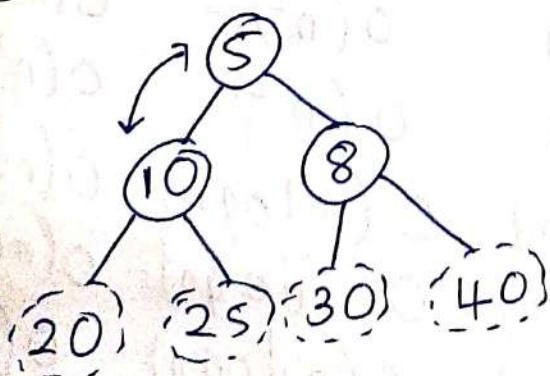
@ 5

swap root with last

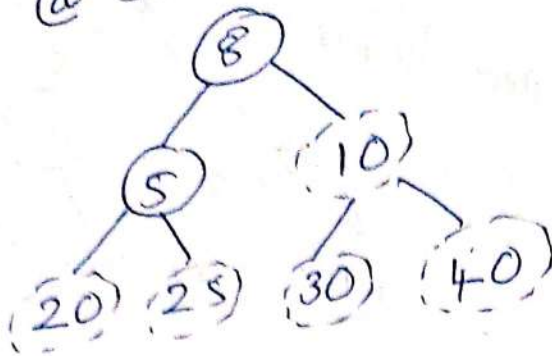


@ 4

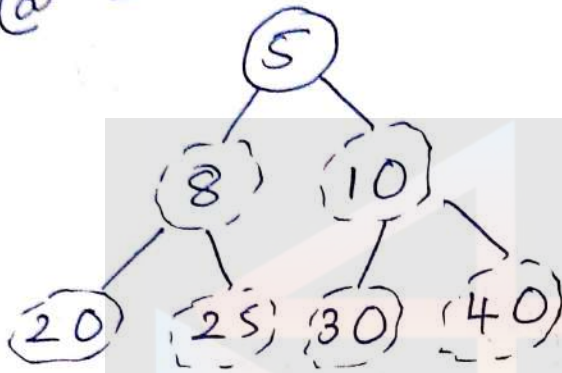
swap root with last



@ 3



@ 2



Hence finally. *Ans*

5	8	10	20	25	30	40
---	---	----	----	----	----	----

Heap sort take $O(n \log n)$ to sort

<u>Sorting Algo:</u>	<u>Best</u>	<u>Average</u>	<u>Worst</u>
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

→ Hashing:

→ refer basic concepts of hashing
(SW DS Unit 2)

→ A dictionary can even be represented by using a method called hashing.

→ hashing is a technique where search time is independent of the number of items in which we are searching data.

(Key value pair)

$$H : K \rightarrow L$$

good hashing:

→ easy to compute & understand

→ less time consumption

→ must have low collision rate

↳ 2 or more
values at
1 key location

→ Set Theory: (a collection of well defined & well distinguished objects).

Set: Set is a collection of object of ~~any~~ ^{particular} kind which are collected together

→ The objects which make up the set are called it's members or elements.

→ IF the no. of elements in the set are finite the set is called finite set.

→ IF set has infinite no. of elements then ^{it's called} infinite set.

→ Set having only one element is called Singleton set.

→ Set is designated by $\{ \}$.

→ set with no elements is called empty set or Null set. → $\{ \}$

$A = \{ 1, 2, 3 \}$ → finite set

$B = \{ 1, 2, 3, \dots \}$ → infinite set

$C = \{ 49 \}$ → singleton set

$D = \{ \}$ → empty set

→ Two methods for describing a set:

i) Tabulation method: All elements of the set are written down within the braces

ex: A set of vowels

$$A = \{a, e, i, o, u\}$$

ii) Rule method:- Characteristic property is specified which all the elements of set possess

ex: $A = \{x : x \text{ is vowel in english alphabets}\}$

→ Types of sets:

- i) Null or empty ✓
- ii) Unit or Singleton set ✓
- iii) Finite set ✓
- iv) Infinite set ✓
- v) Equal set: Two sets A & B are said to be equal if they have precisely the same elements then

$$A = B \text{ (A equals B).}$$

ex

$$\left. \begin{array}{l} A = \{1, 2, 3, 4\} \\ B = \{3, 2, 1, 4\} \end{array} \right\} A = B \checkmark$$

vi) Equivalent Sets: Two sets A & B are said to be equivalent (or) similar sets if they have same cardinality.

ex:
$$\left. \begin{array}{l} A = \{1, 2, 3\} \\ B = \{a, b, c\} \end{array} \right\} |A| = |B| = 3$$

vii) Subset: Given two sets A & B

A is the subset of B or
 A is contained in B

if every element of A
is an element of B

$$A \subseteq B$$

$$A = \{1, 2, 3, 7\}$$

$$B = \{1, 2, 3, 4, 5, 6, 7\}$$

$$A \subseteq B$$

viii) Proper subset: A set A is proper subset of B

if $A \subseteq B$ & B possess

atleast one element that is not in A

$$A \subset B$$

$$\begin{aligned} A &= \{1, 2, 3, 7\} \\ B &= \{1, 2, 4, 3, 6, 7\} \end{aligned} \quad \left. \vphantom{\begin{aligned} A &= \{1, 2, 3, 7\} \\ B &= \{1, 2, 4, 3, 6, 7\} \end{aligned}} \right\} A \subset B$$

ix) Universal set: Any set which is a super set of all the sets under consideration is known as universal set denoted by X, S, U .

ex: $A = \{1, 2\}$

$$B = \{4, 7, 2\}$$

$$C = \{1, 4, 3\}$$

$$U = \{1, 2, 3, 4, 7\}$$

x) Power Set: The family of all the subsets of a given set A is said to be the power set of A .

ex $A = \{1, 2, 3\}$

$$P(A) = \{ \phi, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \}$$

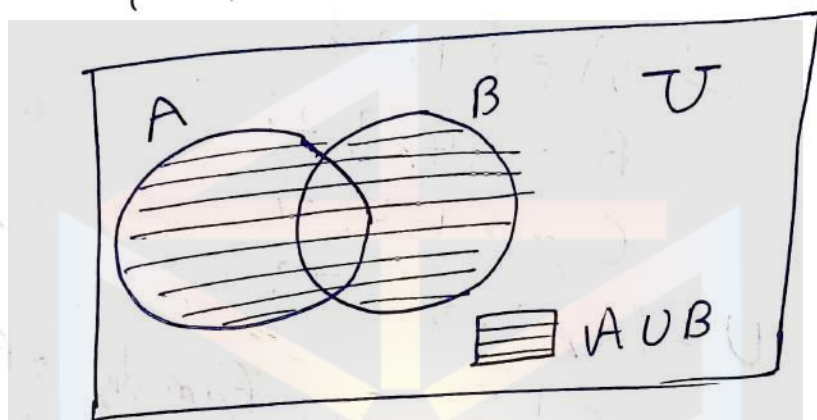
$$P(A) = 2^n \quad (n = \text{no. of elements})$$

* To represent relationship b/n two or more sets we use Venn Diagrams.

→ Union of sets:

Consider two sets A, B
Then the set consisting of all elements
that belong to A or B is called
union of A & B , $(A \cup B)$

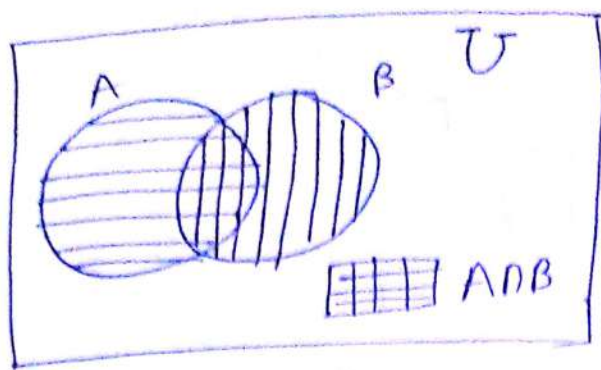
$$A \cup B = \{ x / x \in A \text{ or } x \in B \}$$



→ Intersection of sets:

Consider two sets A, B
Then the set containing of all elements
that belong to both A & B is
called intersection of A & B

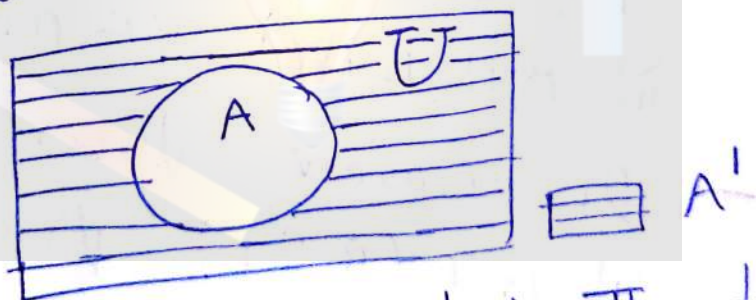
$$A \cap B = \{ x / x \in A \text{ \& } x \in B \}$$



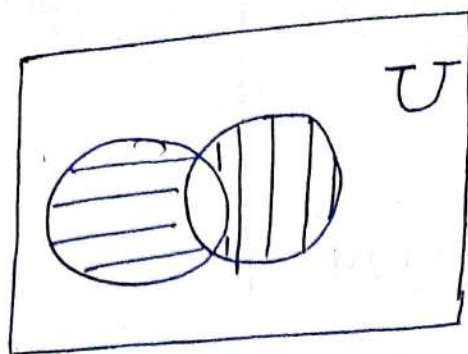
→ Complement of set :



Given a universal set U and set A which is contained in U

The set of elements that belong to U but not to A is called complement of A denoted by A^c or \overline{A} or A'



→ Difference of two sets: The difference of two sets A & B is the set of all those elements which belong to A & not to B denoted by $A - B$



 $A - B$
 $B - A$

→ Laws of set theory:

→ Commutative laws

i) $A \cup B = B \cup A$

ii) $A \cap B = B \cap A$

→ Associative laws

i) $A \cup (B \cap C) = (A \cup B) \cap C$

ii) $A \cap (B \cup C) = (A \cap B) \cup C$

→ Distributive laws

i) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

ii) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

→ Idempotent laws

i) $A \cup A = A$

ii) $A \cap A = A$

→ Identity laws

i) $A \cup \phi = A$

ii) $A \cap U = A$

→ Inverse laws

i) $A \cup \bar{A} = U$

ii) $A \cap \bar{A} = \phi$

→ De Morgan's law

i) $\overline{A \cup B} = \bar{A} \cap \bar{B}$

ii) $\overline{A \cap B} = \bar{A} \cup \bar{B}$

→ Domination law

i) $A \cup U = U$

ii) $A \cap \phi = \phi$

→ Absorption law

i) $A \cup (A \cap B) = A$

ii) $A \cap (A \cup B) = A$

→ Disjoint Set:

A disjoint set is a kind of data structure that contains partitioned sets. These partitioned sets are separate non-overlapping sets.

$\therefore A \cap B = \phi$

ex1 $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

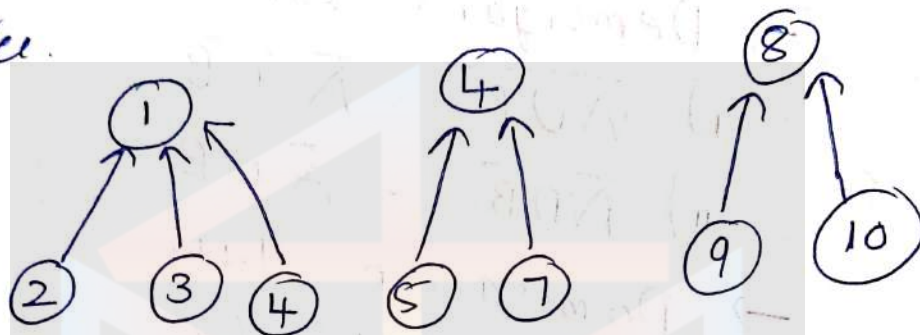
assume disjoint sets

$$A = \{1, 2, 3, 6\}$$

$$B = \{4, 5, 7\}$$

$$C = \{8, 9, 10\}$$

each of the set can be represented as a tree.



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

-1 means its parent
1 means its parent

→ Operations on disjoint set:

i) M/W: Min operation is used to determine the number of elements in the set that contain the min number of elements.

ex1 $P = \{1, 2, 3, 4, 5\}$ & $Q = \{14, 15\}$

$$\min \{P, Q\} = 2$$

since P has 5 elements & ~~Q~~
Q has 2 elements.

ii) Delete: Delete operation is used to delete the given element from the respective set. This is done by finding the set to which the element belongs to.

ex: $P = \{4, 5, 6, 7\}$ and $Q = \{10, 11, 12\}$

delete (6)

= 1st step is to find (6)
assume we did it & got P set as result

now delete (6)

$$\therefore P = \{4, 5, 7\}$$

* iii) Find: Find operation i.e., Find(i) is used to find the set to which the given element 'i' belongs to.

ex: $P = \{1, 2, 3\}$ & $Q = \{4, 5\}$

$$\text{Find}(4) = Q$$

* iv) Union: Union operation is used to combine all the elements of two given sets. It is represented by \cup .

ex: if $P = \{1, 2, 3\}$ & $Q = \{4, 5, 6\}$

$$P \cup Q = \{1, 2, 3, 4, 5, 6\}$$

v) Intersect: Intersect operation is used to determine the common elements of the two given sets. represented by \cap

ex: if $P = \{3, 4, 5\}$ & $Q = \{6, 7\}$

$$P \cap Q = \{\phi\}$$

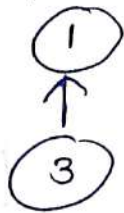
→ Union: Union(i, j) (Simple Union)

it means the elements of set i and elements of set j are combined. If we want to represent Union operation in the form of a tree.

The Union(i, j), i is the parent(new), j is the child(new)

ex:

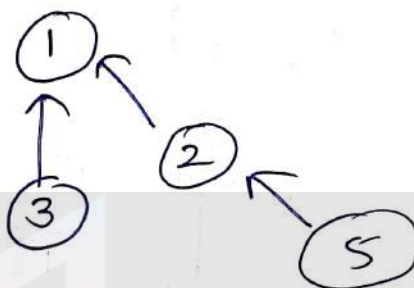
Union (1, 3)



Union (2, 5)



Union (1, 2)



Algorithm :- Union (i, j)

{

// $i \neq j$

integer i, j ;

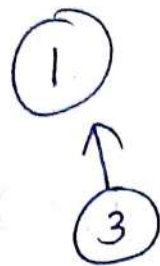
parent array \leftarrow Parent(j) \leftarrow i ;

}

ex: Union (1, 3) , Union (2, 5), Union (1, 2)
initially.

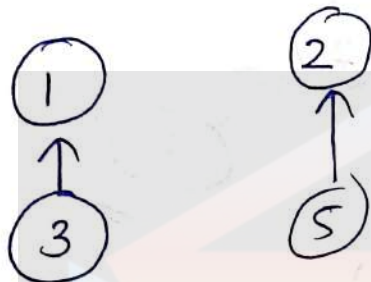
0	0	0	0	0	0
1	2	3	4	5	6

Union (1, 3)



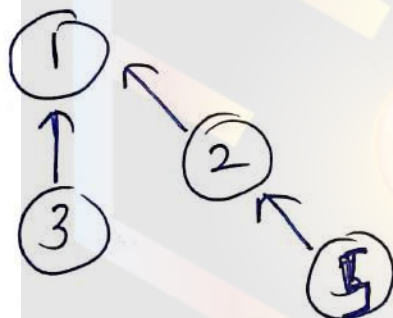
0	0	1	0	0	0
1	2	3	4	5	6

Union (2, 5)



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

Union (1, 2)

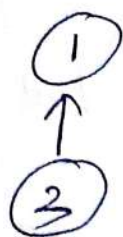


0	1	1	0	2	0
1	2	3	4	5	6

→ Find Operation: (Simple find) =

find(i) implies that it finds the root node of ith node, in other words, it returns the name of the set i.

ex:



$$\text{Find}(1) = 1$$

$\text{Find}(3) = 1$ since its parent is 1 (root node is 1)

A algorithm $\text{find}(i)$:

{ // Find the root for the tree containing element

integer i, j ;

$j \leftarrow i$

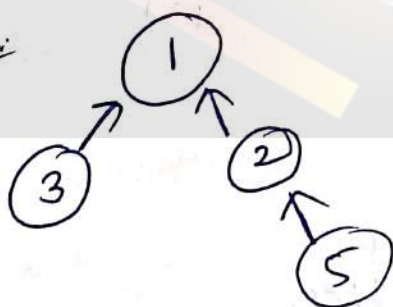
while $\text{Parent}(j) > 0$ do

$j \leftarrow \text{Parent}(j)$

repeat

return j

ex:



Parent

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

$$\text{Find}(5) \quad i = 5, j = 5$$

$$P(5) = 2 \Rightarrow 2 > 0 \text{ do } j = 2$$

$$P(2) = 1 \Rightarrow 1 > 0 \text{ do } j = 1$$

$$P(1) = 0 \Rightarrow 0 > 0 \times$$

$$\text{return } j \Rightarrow \underline{1}$$

1 parent-
or
root

→ issue

ex: {1} {2} {3} {4} ... {n}
single elements set till 1 ... n

i	1	2	3	4	...	n
p	-1	-1	-1	-1		-1

→ as all
are
parents.

Operation are: Union (1, 2) Union (2, 3) ... Union (n-1, n)

Operation are: Find (1), Find (2) ... Find (n)

n

n-1

2

1

So total time
needed =

$$O\left(\sum_{i=1}^n i\right) = O(n^2)$$

Regenerate here.

to resolve this

we follow

below

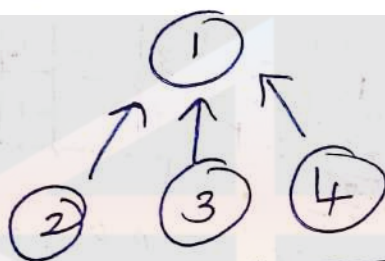
updated algo's

→ Weighting rule for Union(i, j)

→ IF the number of nodes in tree i is less than the number in tree j, then make j as the parent of i, otherwise make i as the parent of j.

→ Store the no. of element in set (-n) in the root parent.

ex)



i	1	2	3	4
p	-4	1	1	1

Algorithm Weighted Union (i, j)

{ // Union set ^{root} i & j, i ≠ j
 // weighting $p[i] = -\text{count}[i]$,
 $p[j] = -\text{count}[j]$.

$\text{temp} \leftarrow \text{Parent}[i] + \text{Parent}[j]$
 if ($\text{Parent}[i] > \text{Parent}[j]$) then
 { // i has fewer nodes
 $p[i] = j$ $p[j] = \text{temp}$.
 }

else

{ // j has fewer nodes

Parent(j) ← i

Parent(i) ← temp

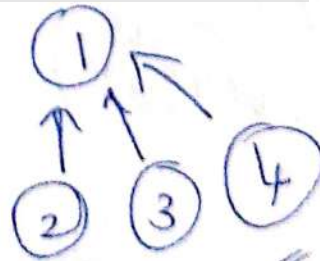
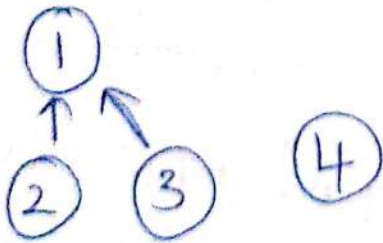
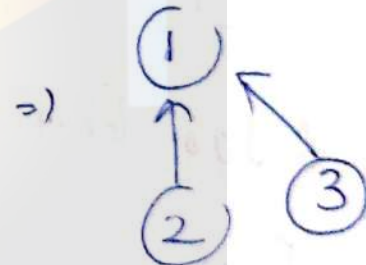
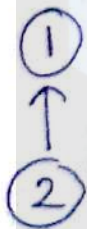
}

ex:

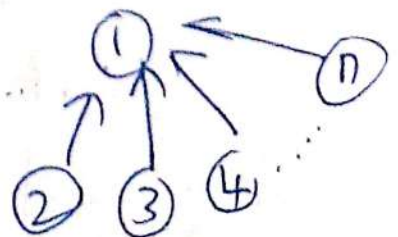
i	1	2	3	...	4
p	1	2	3	...	-1

Union(1,2), Union(2,3), Union(3,4) ... Union(n-1,n)

i	j	temp
1	2	-3



⇒



→ Collapsing Rule For Find(i)

IF j is a node on the path
from i to its root set $\text{Parent}(j)$
 $\leftarrow \text{root}(i)$, & the start collapsing.

Algorithm : Collapsed Find(i)

```
{  
  // Find the root of the tree  
  // containing element- i  
  // Use collapse all nodes from i to  
  // the root  
   $r \leftarrow i$   
  while ( $p[r] > 0$ ) do // Find root-  
     $r = p[r]$   
  while ( $i \neq r$ ) do // collapsing  
  {  
     $s = p[i]$   
     $p[i] = r$   
     $i = s$   
  }  
  return  $s$   
}
```

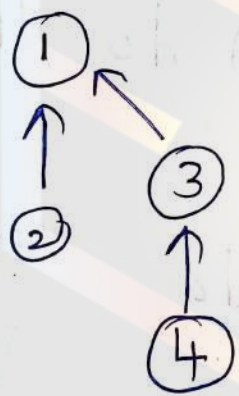
ex: Union(1, 2) Union(3, 4) Union(5, 6)
 Union(7, 8) Union(1, 3) Union(5, 7)
 Union(1, 5)

	-8	1	1	3	1	5	5	1
i	1	2	3	4	5	6	7	8
Parent	1	1	1	1	1	1	1	1

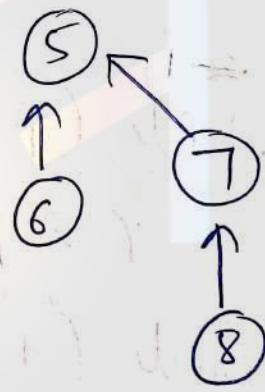
Union(1, 2) Union(3, 4) Union(5, 6) Union(7, 8)



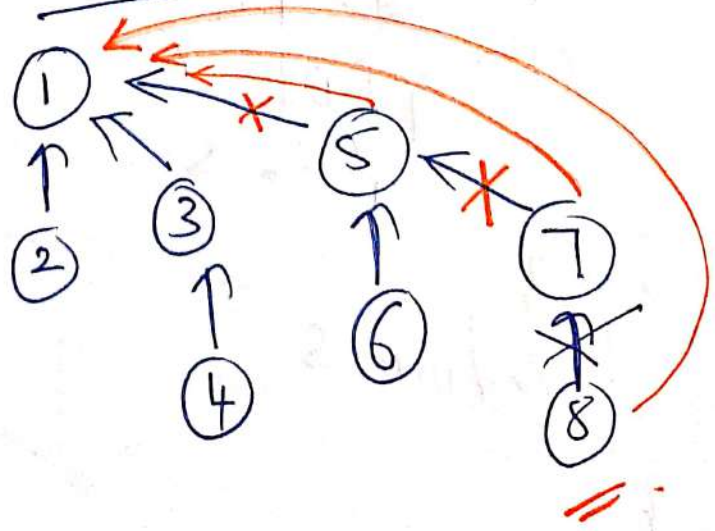
Union(1, 3)



Union(5, 7)



Union(1, 5)



Find (8)

i	r	s
8	8	
	7	
	5	
	1	

(-8 70) x
1 root

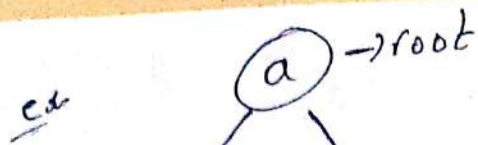


return 1.

Now we find again
5, 7, 8 any 1 we
can get in 1 step
instead of all path.

→ Trees: A tree is a simple graph G such that there is a unique simple non directed path b/n each pair of vertices of G (or)

A connected graph without any circuit is called tree.



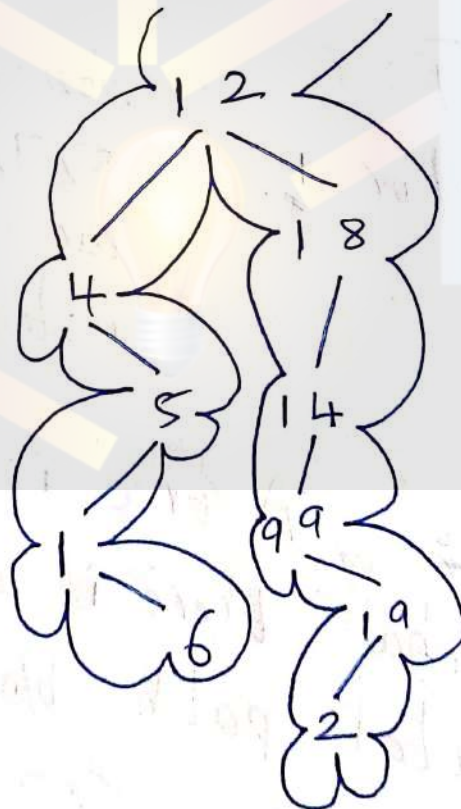
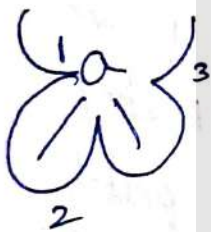
→ dvl 0

→ dvl 1

→ dvl 2

Tree traversal (3 types)

- i) Preorder Traversal (P, L, R)
- ii) Inorder Traversal (L, P, R)
- iii) Postorder Traversal (L, R, P)



Pre order

12, 4, 5, 1, 6, 18,
14, 99, 19, 2.

Preorder (T)

{ if (T ≠ NULL)

{ Visit (T);

Preorder (T → left);

Preorder (T → right); }

}

Post order:

6, 1, 5, 4, 2, 19, 99, 14, 18, 12

Post order (T)

{ if (T ≠ NULL)

{ Post order (T → left);

Post order (T → right);

Visit (T);

}

}

In order

4, 1, 6, 5, 12, 99,
2, 19, 14, 18

Inorder (T)

{ if (T ≠ NULL)

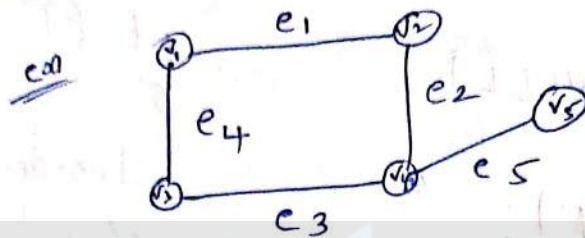
{ Inorder (T → left);

Visit (T);

Inorder (T → right); }

}

→ Graph: A graph G is a pair of set (V, E)
 \downarrow \downarrow
 vertices edges



↳ 5 edges $[e_1, e_2, e_3, e_4, e_5]$

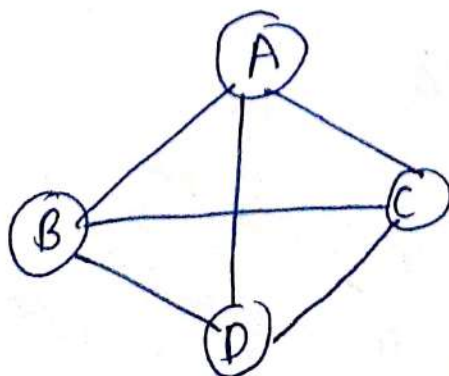
↳ 5 vertices $[v_1, v_2, v_3, v_4, v_5]$.

Basics

→ Undirected Graph: A graphⁱⁿ which the elements of the edge set are unordered pairs of vertices is called Undirected

ex: $V(G) = \{A, B, C, D\}$

$E(G) = \{(A, B), (A, C), (A, D), (B, C), (B, D), (C, D)\}$

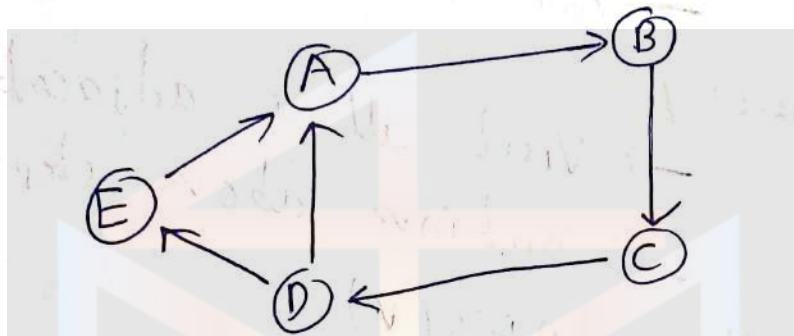


→ Directed Graph: A graph which the elements of the edge set are ordered pair of vertices called directed graph

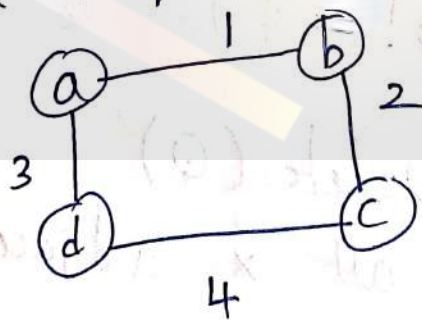
ex

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

$$E(G) = \{(A, B), (B, C), (C, D), (D, A), (D, E), (E, A)\}$$



→ Weighted graph: A graph in which weights are assigned to every edge in a weighted graph.



1, 2, 3, 4 are weights of the graph

→ Graph Traversal:

i) Breadth First Search (BFS)

→ Let $G = (V, E)$ be a connected graph of order n , with vertices v_1, v_2, \dots, v_n in some specified order (Queue).

→ You can select any vertex as starting

→ Visit all x adjacent vertices & continue above steps

algorithm: BFS(v)

{

visited(v) = 1

insert [v, Q]

while ($Q \neq \emptyset$)

{

$u = \text{Delete}(Q)$

For all x adjacent to u

{ if (x is not visited)

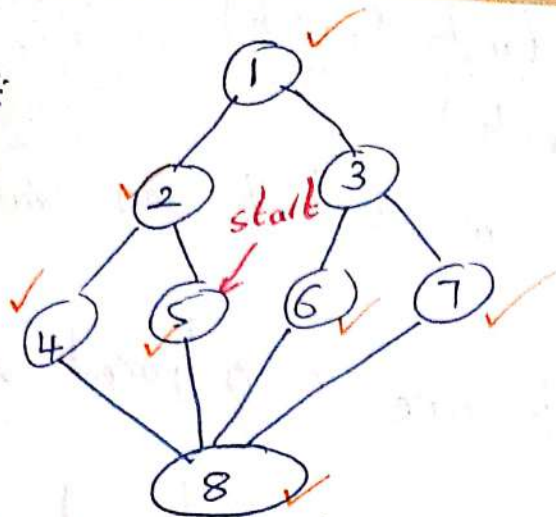
{

visited(x) = 1

insert (x, Q)

} } }

ex:



@ BFS(S)

visited[S] = 1

insert[V, Q]

S Queue

while (Q not empty)

U = S

all adjacent
if not visited

visit & push to queue & continue
until Queue is empty.

Queue has

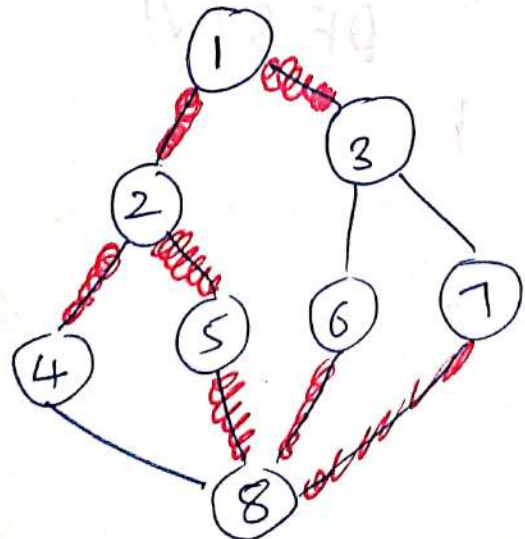
1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0

visited

Queue

S	2	8	1	4	6	7	3
---	---	---	---	---	---	---	---

S	2	8	1	4	6	7	3
---	---	---	---	---	---	---	---



BFS.

ii) DFS (Depth First Search)

→ (Stack)

→ You can select any vertex

a) starting (x)

→ Visit one of adjacent vertex

(y)

→ visit one of adjacent vertex

if not found come back & continue

till we visit all vertices

algorithm: DFS(v)

{ visited(v) = 1

For all x adjacent to v

{ if (x is not visited)

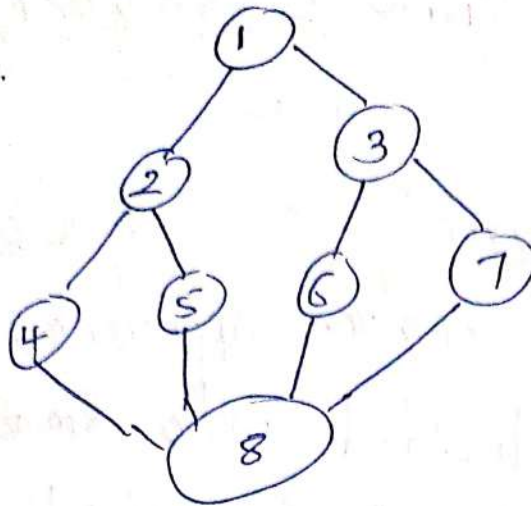
{

DFS(x)

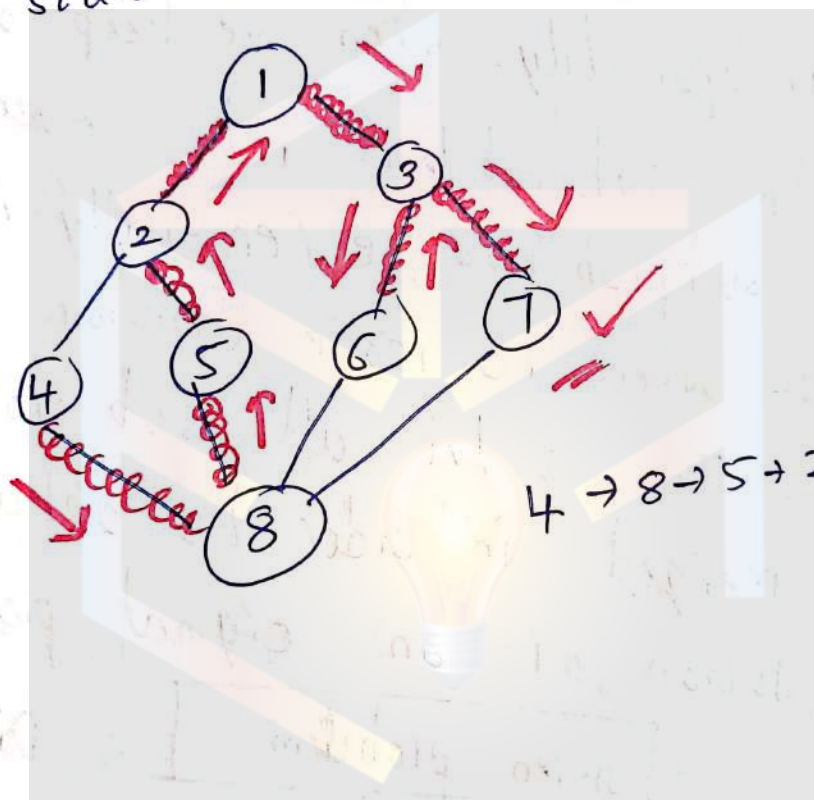
}

}

ex!



start at (4).



4 → 8 → 5 → 2 → 1 → 3 → 6
↓
3
↓
7
=