# UNIT-1

# CHAPTER-1.

# INTRODUCTION TO ALGORITHMS

## Definition of Algorithm-

The name 'Algorithm' is given after the name of "Abu Ja'far Muhammad ibn Musa AL-KHWARIZMI".

The algorithm can be defined as follows:-

① An algorithm is a set of rules for carrying out calculation either by hand or on a machine

② An algorithm is sequence of computational steps that transform the input into the output.

③ A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific problem or a class of problems is called an algorithm.

## Properties of Algorithm:-

An algorithm must have the following properties-

① Finiteness :- Algorithm must complete after a finite number of instructions have been executed.

② Absence of Ambiguity :- Each step must be wel defined, having only one interpretation.

③ Definition of Sequence :- Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.

④ Input / Output :- Number and types of required input and results must be specified.

⑤ Feasibility :- It must be possible to perform each instruction.

## Characteristics of An Algorithm

Each algorithm must have the following characteristics or features :-

1. Input
2. Ouput
3. Definiteness
4. Effectiveness
5. Termination

Input / Output :- According to (1) & (2) characteristic An algorithm can produce One or more outputs and can have zero or more inputs that are externally supplied.

Definiteness :- According to this characteristic each operation must be definite meaning that it must be perfectly clear what should be done.

Effectiveness :- This characteristic requires that each operation be effective that is each step must be such that it can be done by a person using pencil and paper in a finite amount of time.

Termination :- This characteristic of an algorithm is that it terminates after a finite number of operations.

## ALGORITHM DESIGN TECHNIQUES

There are many ways to design an algorithms-

1. Devide and Conquer
2. Dynamic Programming
3. Greedy Approach
4. Randomized Algorithms
5. Branch and Bound
6. Backtracking

1. Divide and conquer.

* Divide the original problem into set of sub problems.
* Solve every sub problem individually, recursively
* combine the solutions of the subproblems into a solution of the whole original problem.

2. Dynamic Programming :- Dynamic Programm Bo
is a technique for efficiently computing
recurrences by storing the partial results. It
is method of solving problems exhibiting the
properties of overlapping sub problems and optimal.
Structure that takes much less time.

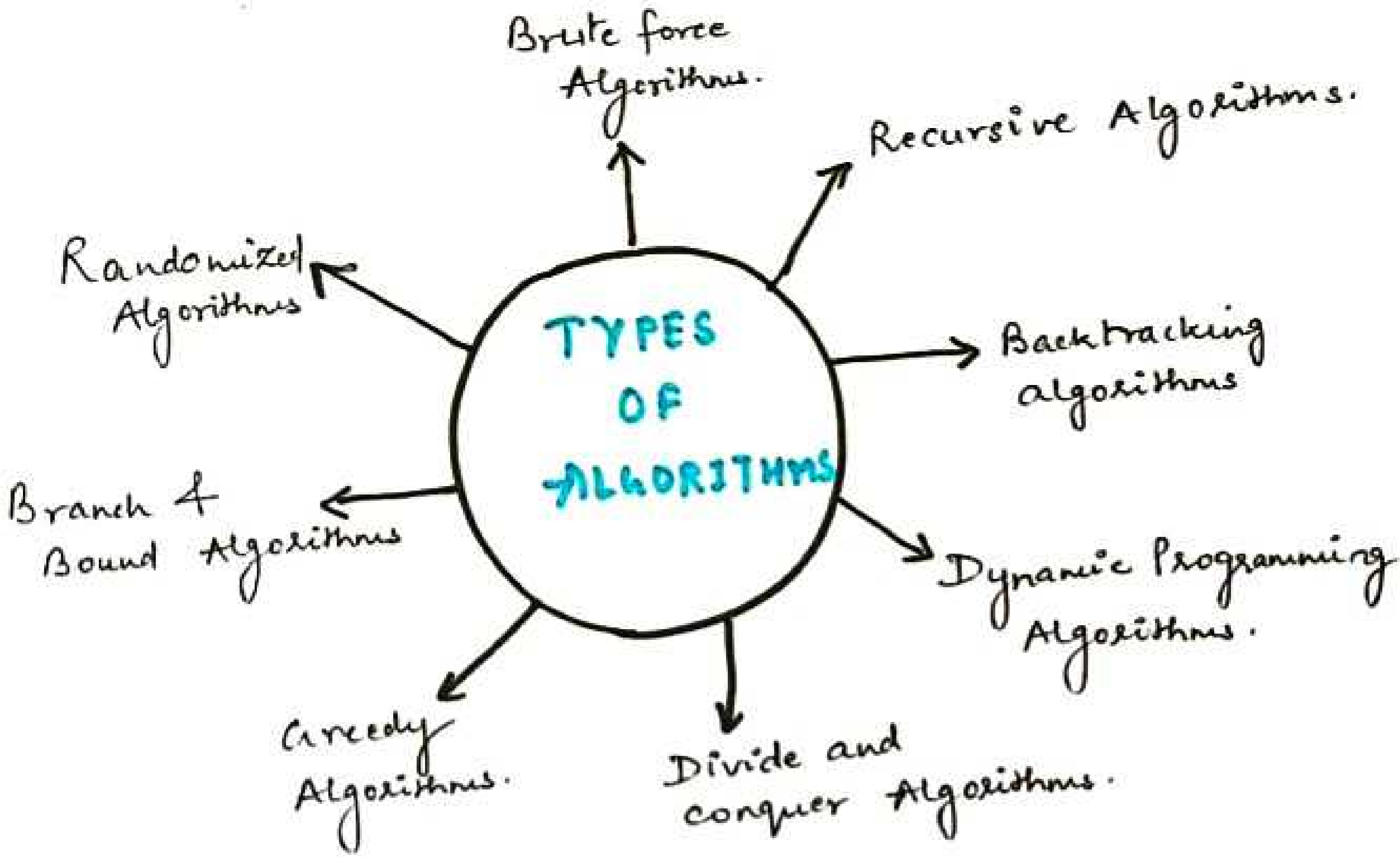
3. Greedy Approach. Greedy algorithm seek to
optimize a function by making choices which are
the best locally but do not look at the global
problem. The result is a good solution but not
necessarily the best one. The greedy algorithm does
not always guarantee the optimal solution however
it generally produces solution that are very close
in value to the optimal.

4. Randomized Algorithms :- A randomized algorith
is defined as an algorithm that is allowed to
access a source of independent, unbiased random
bits, and it is then allowed to use these random
bits to influence its computation.

5. Branch and Bound. In a Branch and
Bound algorithm a given
Sub problem, which cannot be bounded has to be
divided into at least two new restricted subproblems

Backtracking Algorithms:- Backtracking algorithms try each possibility until they find the right one. It is a depth-first search of the set of possible Solutions. During the search, if an alternative. doesnot work, the search backtracks to the choice point, the place which presented different alter-natives and tries the next alternative. If there are no more choice points, the search fails.

## ALGORITHM CLASSIFICATION



## Recursive Algorithms

A recursive algorithm :-

* Solve the base cases directly
* Recurs with a simple sub problem
* Does some extra work to convert the solution to the simpler sub problem into a solution to the given problem

## 2. Backtracking Algorithms.

Backtracking algorithm are based on a depth-first recursive search. A backtracking algorithm:

* Test to see if a solution has been found, and if so, returns it ; otherwise.
* For each choice that can be made at this point,
  (1) Make that choice
  (2) Recur
  (3) If the recursion returns a solution, return it
* If no choice remain, return failure.

## 3. Divide and conquer

A divide and conquer algorithm consists of two parts :-

* Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively.

* combine the solutions to the subproblems into a solution to the original problem.

4. <u>Dynamic Programming Algorithms</u>.

A dynamic programming algorithm remembers past results and uses them to find new results.
Dynamic programming is generally used for optimization problems. In dynamic programming multiple solutions exist, need to find the "best" one.
It "requires substructure" and "overlapping subproblems".
Optimal substructure means optimal solution contains optimal solutions to sub problems. Overlapping subproblems means solutions to sub problems can be stored & reused in a bottom-up fashion.

5. <u>Greedy Algorithm</u>.

An optimization problem is one in which we want to find, not just a solution, but the best solution.
A "greedy algorithm" sometimes work well for optimization problems (A problem having a set of feasible solutions choose the best-one).

A greedy algorithm works in phases :-
At each phase :-

* We take the best we can get right now, with-out regard for future consequences.
* We hope that by choosing a local optimum at each step, we will end $\overset{up}{\wedge}$ at global optimum.

⑥ Branch and Bound Algorithms

Branch and Bound algorithms are generally used for optimization problem. As the algorithm progresses, a tree of subproblems is formed. The original problem considered $\overset{as}{\wedge}$ the "root problem"

A method is used to construct an upper & lower bound for a given problem.

* At each node, apply the bounding methods.
* If the bounds match, it is deemed a feasible solution to that particular sub problem.
* If the bound donot match, partition the problem represented by that node, and make the $\overset{two}{\wedge}$ sub problems into children nodes.

⑦ Brute force Algorithm.

A ~~Bot~~ Brute force algorithm simply tries all possibilities until a satisfactory solution is found. Such an algorithm can be:-

⇒ Optimizing:- Find the best solution from the set of available solutions.

⇒ Satisfying:- stop as soon as solution is found that is good enough.

B) Randomized Algorithms

A randomized algorithm uses a random number at least once during the computation to make a decision.

## ALGORITHM ANALYSIS METRICS.

1. Input size.
2. Running Time :- The running time of an algorithm on a particular input is the number of primitive operations or steps executed.

3. Order of growth :- To simplify the analysis of algorithms, we are interested in growth rate of the running time .i.e we only consider the leading terms of a time formula eg:- the leading term is $n^2$ in the expression $n^2 + 100n + 5000$.

## HOW TO CALCULATE RUNNING TIME OF AN ALGORITHM ?

To calculate the running time of an algorithm we need to consider the following metrics -

1. Basic Operation :- The time to complete a basic operation does not depend on the particular values of its operands. So it takes constant amount of time.

Example :- airthmetic operation (addition, substraction, multiplication etc), Boolean operation, comparision operation, Branch operation etc. ①

2. Input size :- It is the number of input processed by the algorithm.

3. Growth Rate :- The growth rate of an algorithm is the rate at which the running time of the algorithm grows as the size of the input grows.

———x ———x ——— x ———x ———

NOTES

Decision problem :- The Decision problem always having only 2 options.

→ YES

→ NO.

Optimization problem :- Such kind of problem having a set of feasible solutions, from where we chooses the best one.

# UNIT-1

# CHAPTER-2

# GROWTH OF FUNCTIONS

## COMPLEXITY OF ALGORITHMS

The complexity of algorithm depends upon the relative amount of time or the relative amount of space they require and specify the growth of time space requirements as a function of the input size

1. **Space complexity** :- space complexity of an algorithm is the amount of memory it needs to run to the completion, Generally, space needed by an algorithm is the sum of following two components :

1. A fixed part that is independent of the characteristics of the inputs and outputs. This part includes instruction space (ie, space for the code), space simple variables and fixed size component variables, space for constants and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance.

RAM Model:

referenced variable

being solved, the space needed by referenced variable & the recursion stack space

Thus, the space required by any problem x can be computed as ,

where $S(x) \rightarrow$ space required by a problem x

$c \rightarrow$ is a constant

$S_x \rightarrow$ is an instance

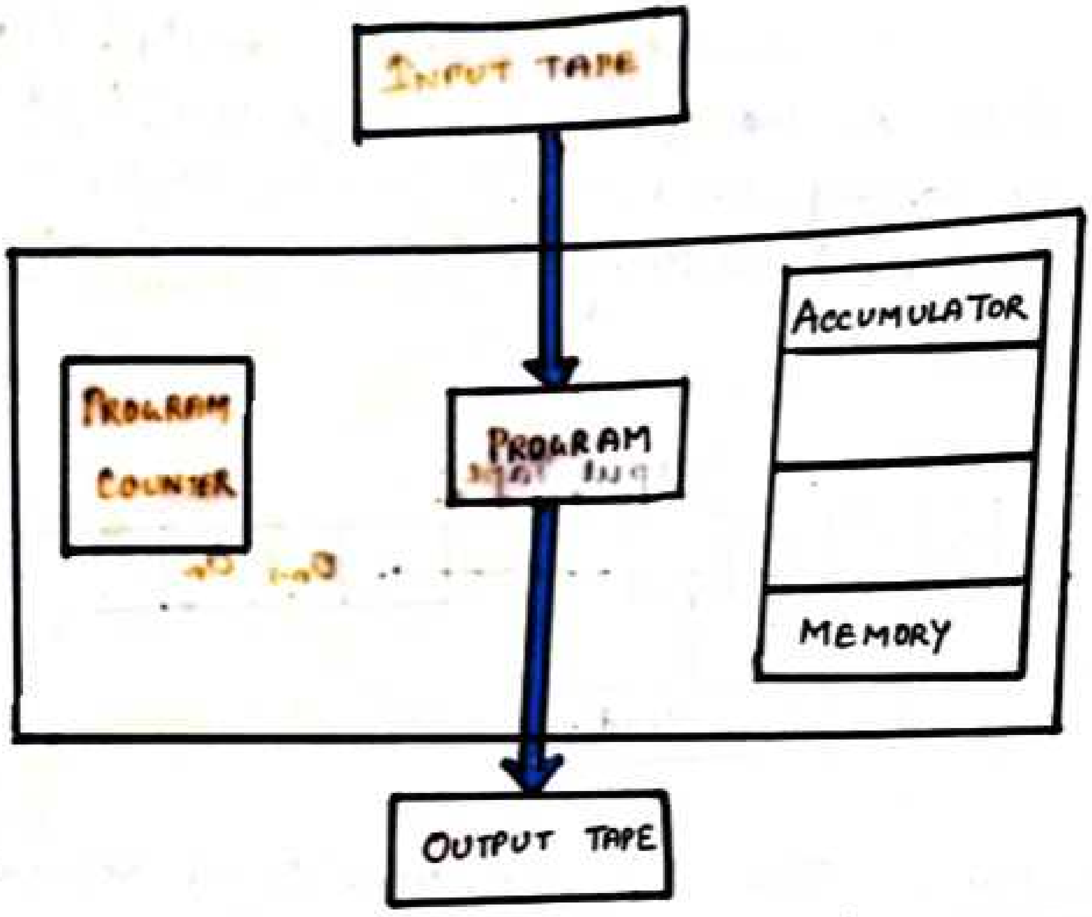
Instance characteristic ($S_x$) varies from problem to problem, so during analyzing the space complexity we concentrate solely on estimating $S_x$

2. Time Complexity.

Time complexity of an algorithm is the amount of computer time it needs to run to completion. This is the sum of compile time and run time. The compile time does not depend on the instance characteristics and a program compiled once can be executed many times without recompiling it.

For calculating the run time of a program we have to calculate time taken in addition, multiplication, substraction, division, load and store and so on. We count the number of operations. We can count the number of program steps and the number of steps of any program depends on the kind of statement.

## RAM Model:-



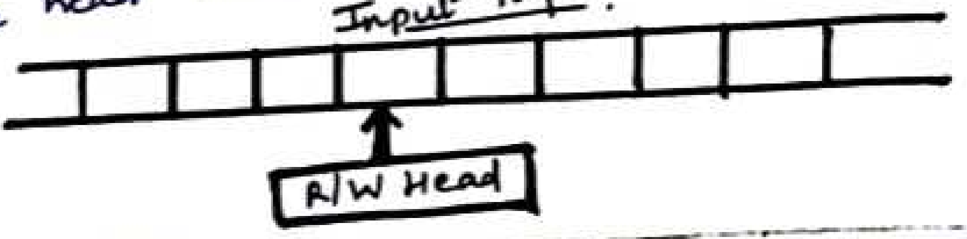
BLOCK DIAGRAM OF RAM MODEL.

The RAM model consists of following elements

1.) Input tape
2.) Output tape
3.) A Program.
4.) A Memory
5.) Program counter.

1. Input Tape :- The input tape consist of sequence of squares, each of which can store in integer whenever one square is read from the tape, then the head moves one square to the right.

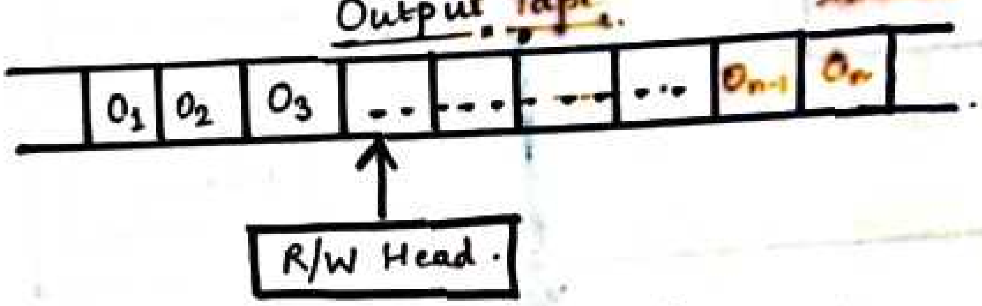
Input Tape.



2.) Output Tape.

integer c[illegible]

is written [illegible] the [illegible]
after the writing, the t[illegible]
the right, over writing
not permitted.



3) Memory:- The memory consists of sequence of registers, each of which is capable of holding an integer or an arbitrary size input.

4.) Program:- A program for RAM contains a Sequence of labelled instructions resembling those found in assembly language programs. All computations take place in the first register called accumulator.

5) Program counter:- The program counter determines the next instruction to be executed.

## BEST, WORST AND AVERAGE-CASE COMPLEXITY

To understand the notions of best, worst and average-case complexity, one must think about the running an algorithm on all possible instances of data that can be fed to it.

The best, worst and average cases of a given algorithm express what the resource usage is at least, atmost and on average, respectively. Usually the resource being considered is running time, but it could also be memory or other resource.

1. Worst case complexity :- The worst case complexity of ~~the~~ an algorithm is the function defined by the maximum number of steps taken on any instance of size n.

2. Average case complexity :- The average case complexity of ~~the~~ an algorithm is the function defined by the average number of steps taken on any instance of size n.

3. Best case complexity :- The best case complexity of ~~the~~ an algorithm is the function defined by the minimum number of steps taken on any instance of size n.

# ASYMPTOTIC NOTATIONS.

Asymptotic notation is a shorthand way to write down and talk about "fastest possible" and "slowest possible" running times for an algorithm, using high & low bounds on speed. They are also referred to as 'best case' & 'worst case' scenarios respectively.

## Importance of Asymptotic Notations

1. They give a simple characterization of Algorithms efficiency.
2. They not only allow to measure but also compare the performance of various algorithms.

## Asymptotic Notations are of 5 types.

1. Big-oh Notation (O):- Big-oh notation is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could be possibly taken by an algorithm to complete.

More formally, for the non-negative function, $f(n)$ & $g(n)$, if there exists an integer $n_0$ and a constant $c$ $(c>0)$ such that for all integers $n>n_0$ ·

$\boxed{f(n) \leq c \cdot g(n)}$

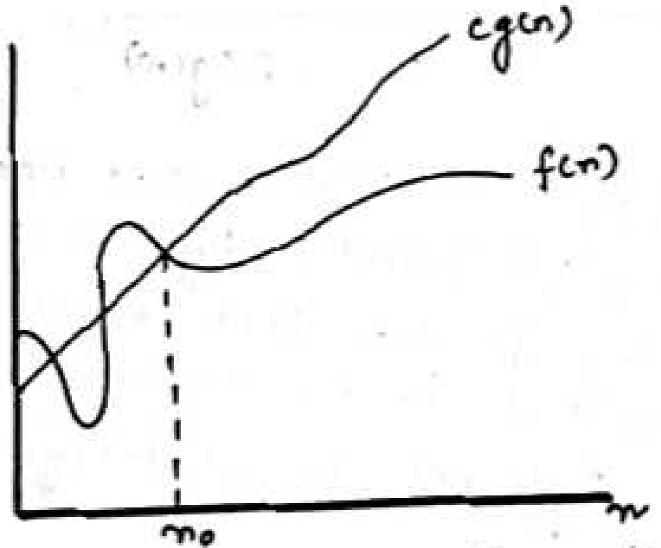
where $n \rightarrow$ Input size.

$f(n) \rightarrow$ function for Input size $n$ which representing time, if the input size increases the value of $f(n)$ also increases.

$g(n) \rightarrow$ Another function for Input size $n$. (~~nf~~ which provides an upper bound for function $f(n)$ $(n \geq n_0)$

Then $f(n)$ is big-oh of $g(n)$. This is denoted as

$$\boxed{f(n) \in O(g(n))}$$

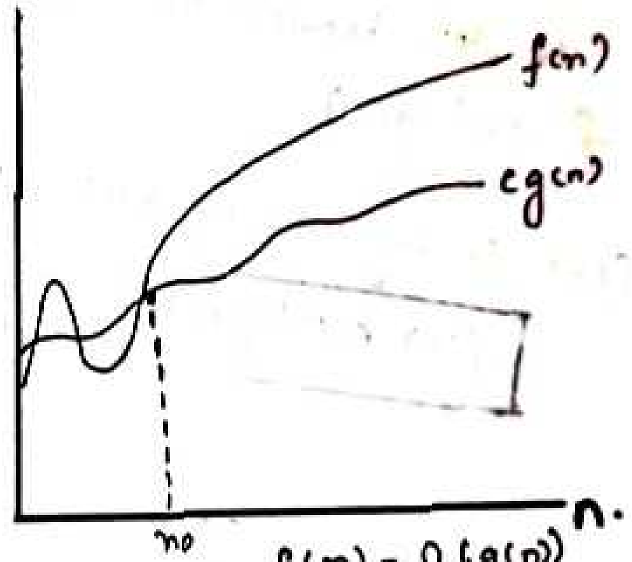


$f(n) = O(g(n))$

2. <u>Big-Omega notation ($\Omega$)</u>:- For non-negative functions, $f(n)$ and $g(n)$, if there exists an integer $n > n_0$, $f(n) \geq cg(n)$. Then $f(n)$ is Big-omega of $g(n)$, this is denoted as.

$\boxed{f(n) \in \Omega(g(n))}$

This is almost the same definition as Big-oh except that " $f(n) \geq g(n)$ " this makes $g(n)$ a lower bound function instead of an upper bound function. It describes the best that can happen for the given data size.



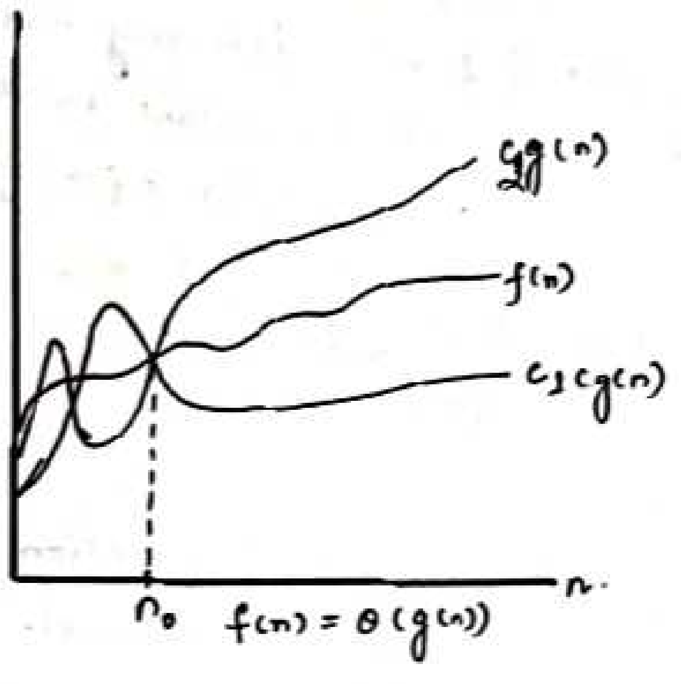
$f(n) = \Omega(g(n))$

3) Thetā, Notation ($\Theta$). The lower and upper bound for the function 'f' is provided by the thetā notation ($\Theta$). For non negative functions $f(n)$ & $g(n)$ if there exists an integer $n_0$ and positive constants $c_1$ & $c_2$ i.e $c_1 > 0$ and $c_2 > 0$ such that for all integers $n > n_0$.

$\boxed{c_1 g(n) \leq f(n) \leq c_2 g(n)}$

Then $f(n)$ is thetā of $g(n)$. This is denoted as $\boxed{\text{"} f(n) \in \Theta(g(n)) \text{"}}$

The thetā notation is the measure of average amount of time it could possibly take for the algorithm to complete.



4. <u>Little-oh Notation</u> (o) :- Asymptotic upper bound provided by O-notation may not be asymptotically tight. So o-notation is used to denote an upper bound that is asymptotically tight.

$$o(g(n)) = \{ f(n) : \text{for any +ve constant } c>0 \text{, if a constant } n_0>0 \text{ such that } 0 \le f(n) < cg(n) \ \forall n \ge n_0 . \}$$

The function f(n) becomes insignificant relative to g(n) as n approaches to infinity i.e

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

5) <u>Little - Omega Notation</u> (ω):- As o-notation is to O notation, we have ω-notation as Ω-notation little-Omega(ω) is used to denote an lower bound that is asymptotically tight.

$\omega(g(n)) = \{ f(n)$: for any +ve constant c > 0 if a constant $n_0 > 0$ such that

$0 \le c\,g(n) < f(n) \}$

Here $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = \infty$

<u>Example 1:-</u> find the O-notation for the following functions:-

(a) $f(n) = 5n^3 + n^2 + 6n + 2$

for $n \geq 2$

$$5n^3 + n^2 + 6n + 2 \le 5n^3 + n^2 + 6n + n$$
$$\le 5n^3 + n^2 + 7n$$

For $n^2 \geq 7n$.

$$5n^3 + n^2 + 7n \le 5n^3 + n^2 + n^2$$
$$\le 5n^3 + 2n^2$$

for $n^3 \geq 2n^2$

$$5n^3 + 2n^2 \le 5n^3 + n^3$$
$$\le 6n^3$$

$c_0 = 6 \quad n_0 = 2$

$$\boxed{f(n) = O(n^3)}$$

(b) $4n^3 + 2n + 3$

$f(n) \le c\,g(n)$

For $n \geq 3$

$$4n^3 + 2n + 3 \le 4n^3 + 2n + n.$$
$$\le 4n^3 + 3n.$$

For $n^3 \geq 3n$.

$4n^3 + 3n \leq 4n^3 + n^3$

$\leq 5n^3$.

$c = 5 \quad n_0 = 3$

$\boxed{f(n) = O(n^3)}$

(c) $f(n) = 10n^2 + 7$.

for $n^2 \geqslant 7$

$10n^2 + 7 \leq 10n^2 + n^2$

$\leq 11n^2$

$c = 11 \quad n_0 = 7$

$\boxed{f(n) = O(n^2)}$

(d) $f(n) = 2^n + 6n^2 + 3n$.

for $n^2 \geqslant 3n$

$2^n + 6n^2 + 3n \leq 2^n + 6n^2 + n^2$

$\leq 2^n + 7n^2$

$n^2 \geqslant 3n$

$n \geqslant 3$

$n \geqslant n_0$

$n_0 = 3$

for $2^n \geqslant n^2$

$2^n + 7n^2 \leq 2^n + 7 \cdot 2^n$

$\leq 8 \cdot 2^n$

$n_0 = 3 \quad c = 8$

$\boxed{f(n) = O(2^n)}$

example 2:- By applying theorem, find out the O-notation for the function $f(n) = 6n^3 + 2n + 6$

Solution:- Given $f(n) = 6n^3 + 2n + 6$

Here $m = 3$ (order of the polynomial)

By theorem, $f(n) = O(n^m) = O(n^3)$

Theorem 1:- If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots - a_1 n + a_0$ and $a_m > 0$ then $f(n) = O(n^m)$

Proof:-

$$f(n) \leq \sum_{k=0}^{m} |a_k| n^k$$

$$\leq n^m \sum_{k=0}^{m} |a_k| \, n^{k-m} \qquad \text{for } n \geq 1$$

$$\leq n^m \sum_{k=0}^{m} |a_k|$$

let

$$|a_m| + |a_{m-1}| + \dots + |a_1| + |a_0| = c$$

$$f(n) \leq c \cdot n^m \qquad \boxed{f(n) = O(n^m)}$$

Theorm 2:- If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$ then show that $f(n) = \Omega(n^m)$

Proof:- $f(n) = a_m n^m + \dots - + a_1 n + a_0$

$f(n) \geq a_m n^m \ \forall\ n \geq 1$

let $a_m = c$ (constant), so $f(n) \geq c \cdot n^m$

$$\boxed{f(n) = \Omega(n^m)}$$

Thm 3:- If $f(n) = a_0 + a_1 n + a_2 n^2 + \dots + a_m n^m$ is any polynomial of degree m or less then

$f(n) = \theta(n^m)$

Proof we know that $f(n) = O(n^m)$ —— (i)
$f(n) = \Omega(n^m)$ —— (ii)

and from (i) & (ii), we can say

$\boxed{f(n) = \theta(n^m)}$

example 3:- Show that $27n^2 + 16n + 25 = \Omega(n^2)$

Solution:- Let $f(n) = 27n^2 + 16n + 25$

$27n^2 \le 27n^2 + 16n + 25 \;\; \forall n.$

$cg(n) \le f(n)$

$c = 27 \quad g(n) = n^2$ $\boxed{f(n) = \Omega(n^2)}$

example 4:- find the $\Omega$ notation for the following functions:-

(a) $5n^3 + n^2 + 3n + 2$

$5n^3 \le 5n^3 + n^2 + 3n + 2 \;\; \forall n.$

$5n^3 \le f(n)$

$cg(n) \le f(n)$

$c = 5 \quad g(n) = n^3$ $\boxed{f(n) = \Omega(n^3)}$

(b) $3^n + 6n^2 + 3n$

$3^n \le 3^n + 6n^2 + 3n$ $\because \{3^n$ having an highest value$\}$

$3^n \le f(n)$

$cg(n) \le f(n)$ $f(n) = \Omega(3^n)$

$c = 1 \quad g(n) = 3^n$

(c) $f(n) = 4 * 2^n + 3n$.

$4 \cdot 2^n \leq 4 * 2^n + 3n$.

$\boxed{f(n) = \Omega(2^n)}$

$c\,g(n) \leq f(n)$ $\quad c = 4$

$g(n) = 2^n$

__example 5__ For the function $f(n) = 27n^2 + 16$ find $\theta$ notation

__Solution__ $f(n) = 27n^2 + 16$.

Now first find lower bound for $f(n)$

$27n^2 \leq 27n^2 + 16$

$c\,g(n) \leq f(n)$

$c_1 = 27 \quad g(n) = n^2$

Now we find upper bound for $f(n)$

for $n^2 \geq 16$ $\quad 27n^2 + 16 \leq 27n^2 + n^2$

$n \geq 4$. $\quad \leq 28n^2$

$c_2 = 28 \quad g(n) = n^2$

$\boxed{f(n) = \theta(n^2)}$ $\quad c_1 = 27 \quad c_2 = 28 \quad n_0 = 4$.

__Example 6__ Find $\theta$-notation for the given exponential function

$f(n) = 3 * 2^n + 4n^2 + 5n + 2$

Now first find lower bound for $f(n)$

$3 * 2^n \leq 3 * 2^n + 4n^2 + 5n + 2 \quad \forall n \geq n_0$.

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

$c_1 = 3 \quad g(n) = 2^n$. $\quad f(n) = \Omega(2^n)$

· example Prove $n! = o(n^n)$

Solution $n! = n(n-1)(n-2) \dots 2.1$
we know that

$= n^n \left[1 - \frac{1}{n}\right] \dots \frac{2}{n} \cdot \frac{1}{n}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^n \left[1 - 1/n\right] \dots \frac{2}{n} \frac{1}{n}}{n^n} = 0$$

$$\boxed{n! = o(n^n)}$$

example Prove $n! = \omega(2^n)$

Solution $n! = n(n-1)(n-2) \dots 2.1$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^n [1-1/n][1-2/n] \dots 2/n \ 1/n}{2^n}$$

$$= \frac{\infty}{2^\infty} = \infty$$

Hence $n! = \omega(2^n)$

example Prove that $f(n) = \log_2 n$ is $O(n^\alpha)$ for any $\alpha > 0$

$g(n) = n^\alpha \qquad f(n) = \log_2 n.$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{\log_2 n}{n^\alpha} = \frac{\log_2 n}{\log 2\, n^\alpha}$$

$$\frac{f'(n)}{g'(n)} = \frac{1/n}{\lg 2 \cdot \alpha n^{\alpha-1}} = \frac{n}{n\alpha \log 2\, n^\alpha} = \frac{1}{\alpha \cdot \log 2\, n^\alpha}.$$

$$= \frac{1}{\infty} = 0$$

Now we find upper bound for the given function

$$3*2^n + 4n^2 + 5n + 2 \le 3*2^n + 4n^2 + 5n + n$$

$$\le 3*2^n + 4n^2 + 6n \qquad \left\{\begin{array}{l} n^2 \ge 6n \\ n \ge 6. \end{array}\right.$$

$$\le 3*2^n + 4n^2 + n^2$$

$$\le 3*2^n + 5n^2 \qquad \{2^n \ge n^2\}$$

$$\le 3*2^n + 5*2^n.$$

$$\le 8*2^n.$$

$f(n) \le c.g(n)$

$c_2 = 8 \quad g(n) = 2^n. \qquad f(n) = O(2^n)$

$$\boxed{f(n) = \Theta(2^n)}$$

Properties

Let $f(n)$ and $g(n)$ be such that $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)}$ exists, then

(i) Function $f \in O(g)$ i.e $f(n) \in O(g(n))$ if

$\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c < \infty$, also including the case in which limit is 0.

(ii) Function $f(n) \in \Omega(g(n))$ if

$\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} > 0$, including the case in which limit is $\infty$

(iii) Function $f(n) \in \Theta(g(n))$ if

$\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c$ for some constant $c$ such that $0 < c < \infty$

**Ques:-** Show that the following equations are correct :

(a) $33n^3 + 4n^2 = \Omega(n^2)$

$f(n) = 33n^3 + 4n^2$

$g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{33n^3+4n^2}{n^2} = n^2(33n+4n)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 33n+4 = \infty$$

Hence $33n^3+4n^2 = \Omega(n^2)$ is correct.

(b) $n! = O(n^n)$

$f(n) = n!$ $\qquad g(n) = n^n.$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{\cancel{n^n}\left(\left[1-\frac{1}{n}\right]\left[1-\frac{2}{n}\right] \cdots \frac{3}{n}\frac{2}{n}\frac{1}{n}\right)}{\cancel{n^n}}$$

$$= \left(\left(1-\frac{1}{\infty}\right)\left(1-\frac{2}{\infty}\right) \cdots \frac{3}{\infty}\frac{2}{\infty}\frac{1}{\infty}\right)$$

$$= 0$$

Hence $n! = O(n^n)$ is correct.

As we know $f(n) = O(g(n))$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$

where $0 \le c < \infty$, i.e including the case in which limit is 0.

(c) $10n^2+9 = O(n^2)$

As we know $f(n) = O(g(n))$

If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c < \infty$

where $0 \le c < \infty$, i.e including the case in which limit is 0.

$f(n) = 10n^2+9$ $\qquad g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{10n^2+9}{n^2} = \frac{\cancel{n^2}(10+ 9/n^2)}{\cancel{n^2}} = 10$$

Hence $10n^2+9 = O(n^2)$ is correct.

(d) $\dfrac{6n^3}{(\log n+1)} = O(n^3)$

Here $f(n) = \dfrac{6n^3}{\log n+1}$ $\qquad g(n) = n^3$

$$\lim_{n\to\infty} \frac{\frac{6n^3}{\log n+1}}{n^3} = \frac{6\cancel{n^3}}{\cancel{n^3}(\log n+1)}$$

$$\lim_{n\to\infty} = \frac{6}{(\log n+1)} = \frac{6}{\infty} = 0$$

Hence $\dfrac{6n^3}{\log n+1} = O(n^3)$ is correct.

Example $\{f_1, f_2, f_3 - - -\}$ is an infinite family of functions such that each function $f_k$ is a function from Natural numbers to natural numbers such that $f_k(n) = O(n)$ then

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

Solution Here $f_k(n) = O(n)$ the we can say that

$f_k(n) \le c.n$

$\sum_{i=1}^{n} f_i(n) = f_1(n) + f_2(n) + \dots + f_n(n)$

Total

$\leq c.n + c.n + cn + \dots + cn$

$\leq c(n + n + n \dots + n)$

$\leq c.n.n \leq cn^2$

$= O(n^2)$

## ASYMPTOTIC NOTATION PROPERTIES

1. Reflexivity

$f(n) = \Theta(f(n))$ $\quad$ $f(n) = \Omega(f(n))$

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

2. Symmetry

$f(n) = \Theta(g(n))$ if and only if

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

3. Transpose Symmetry.

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

4. Transitivity

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$

$f(n) = o(g(n))$ " $g(n) = o(h(n))$ " $f(n) = o(h(n))$

$f(n) = \omega(g(n))$ " $g(n) = \omega(h(n))$ " $f(n) = \omega(h(n))$

## Numaricals Related to Iterative Algorithms.

example :- Consider following 'for' loops, calculate the total computation time for the following

```
for i = 2 to m-1
{
    for j = 3 to i
    {
        sum ← sum + A[i][j]
    }
}
```

i = 2, j = 3 to 2. 0 times | i = 3, j = 3 to 3, 1 time | i = 4, j = 3 to 4, 2 times

i = m-1, j = 3 to m-1, (m-1) times

Solution. The total computation time is

$$\sum_{i=2}^{m-1} \sum_{j=3}^{i} tj = \sum_{i=2}^{m-1} \sum_{j=3}^{i} \theta(1)$$

$$= \sum_{i=2}^{m-1} \theta(i) = \theta\left(\sum_{i=2}^{m-1} i\right)$$

$$= \theta\left(\frac{m^2}{2} + \frac{m}{2} - 1\right)$$

$$= \theta(m^2)$$

$(m-1)(m-1)$

$: m^2 \underline{+} 2m+1$

$: \underline{O(m^2)}$

$2 + 3 + \ldots m-1$

$= \frac{n}{2}[2a + (n-1)d]$ formula.

$= \frac{m-1}{2}[2 \times 2 + (m-2) \times 1]$

example :- For the following program give Big oh analysis of the running time.

(i)
```
for(i=0; i<n; i++)
    A[i] = t;
```

Solution
```
for(i=0; i<n; i++) (n+1) times.
    A[i] = t;
```

```
for(i=0; i<n; i++)   —(n+1) times
{
    A[i] = t;   —— O(n) times.
```

Total running time

$$= O(n) + O(n)$$
$$= 2O(n) \neq O(n)$$

(ii)
```
for( i=0 ; i< n ; i++)
{ for (j=i ; j<n ; j++)
    { for(k=j ; k<n ; k++)
        { S++;
        }
    }
}
```

Solution Our main motive is to calculate how many times this statement (S++) will get executed.

First "for" loop executed

$(n+1)$ times $\rightarrow O(n)$

Second "for" loop executed

$n$ times $\rightarrow O(n)$

third "for" loop executed

$n$ times $\rightarrow O(n)$

So total running time or number of times "S++" this Statement will get executed

$$= O(n) * O(n) * O(n)$$
$$= O(n^3).$$

example:

```
for (i=0; i< n*n ; i++)
    A[i] = i ;
```

Solution

For loop heading will get executed

$(n^2+1)$ times $= O(n^2)$

for loop body will get executed

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

Total running time.

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

$= 2O(n^2) \quad = O(n^2)$

example:- f(n) and g(n) be asymptotically positive functions. write true or false against each of the following

(i) $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$

(ii) $f(n) = \Theta(f(n))$

(iii) $f(n) - O(f(n)) = \Theta(f(n))$

(iv) $f(n) - g(n) = \Theta(\max(f(n), g(n)))$

Solution

(i) $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$

This is the property of Transpose symmetry. This is true.

(ii) $f(n) = \theta(f(n))$

This is the property of Reflexivity. It is True.

(iii) $f(n) - o(f(n)) = \theta(f(n))$ is (False)

Sol -

$f(n) = n^2 + 15n$

$o(f(n)) = n^2$

$f(n) - o(f(n)) = n^2 + 15n - n^2$

$= 15n$

$\neq \theta(n^2)$

$f(n) - o(f(n)) = \theta(f(n))$ is false.

(iv) $f(n) - g(n) = \theta(\max(f(n), g(n)))$

$f(n) = n^2 + 15n \qquad g(n) = n^2$

$f(n) - g(n) = 15n.$

$\max(f(n), g(n)) = f(n)$

$= n^2 + 15n$

$\theta(\max(f(n), g(n))) = \theta(f(n))$

$= \theta(n^2)$

$f(n) - g(n) \neq \theta(\max(f(n), g(n)))$ is false.

us :- Obtain the running time for the following "for" loops:
(a)
```
for(i ← 1 to k)
  {
    for( j ← 1 to k²)
      {
        for( i ← 1 to k³)
          {
             p ← p*p;
          }
      }
  }
```

The running time for executing the statement "p←p*p" is $O(k^5)$ Ans.

Ques: (b)
```
for ( i ← 2 to k - 1)
  {
    for ( j ← 3 to k - 2)
      { A ← A+2;
      }
  }
```

The total running time for executing the statement "A ← A+2" is approx $O(k^2)$ Ans.

# UNIT-1

# CHAPTER-3

# RECURRENCES

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

The methods for solving the recurrences are-

(i) Substitution Method

(ii) Iteration Method

(iii) Master Method

Substitution Method :- It involves guessing the form of the solution and then using mathematical induction to find the constants and show that solution works.

This method is powerful, but it can be applied only in cases when it is easy to guess the form of the answer.

Thus, the substitution method consist of two main steps:

1. Guess the solution
2. Use the mathematical induction to find the boundary conditions and show that the guess is correct.

Example 1:- Solve the recurrence

$T(n) = 2T(n/2) + n$ by substitution method.

Solution:- Guess the solution

$$T(n) = O(n\log n)$$
$$T(n) \leq cn\log n$$

$$T(n) \leq 2T\left(\lfloor n/2 \rfloor\right) + n$$
$$\leq 2\left(c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor\right) + n$$
$$\leq cn\log\lfloor n/2 \rfloor + n$$
$$\leq cn\log n - cn\log 2 + n.$$
$$\leq cn\log n - cn + n.$$
$$\leq cn\log n \quad \forall\ c \geq 1$$
$$\boxed{T(n) = O(n\log n)}$$

example 2:- Consider the recurrence

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$

we have to show that is asymptotically bound is $O(\log n)$

Solution

$$T(n) = O(\log n)$$
$$T(n) \leq c.\log n.$$

Put this in the given recurrence equation

$$T(n) \leq c\log\left(\lfloor n/2 \rfloor\right) + 1$$
$$\leq c\cdot\log(n/2) + 1$$

$\leq c \log n - c \log_2 2 + 1$

$\leq c \log n$ for $c \geq 1$

Thus $T(n) = O(\log n)$

Example 3:- Consider the recurrence

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor + 16\right) + n$$

we have to show that it is asymptotically bound by $O(n \log n)$

Solution $T(n) = O(n \log n)$

$T(n) \leq c\, n \log n$

Put this in the given recurrence equation

$$T(n) \leq 2\left[c\left(\left\lfloor \frac{n}{2} \right\rfloor + 16\right) \log\left(\left\lfloor \frac{n}{2} \right\rfloor + 16\right)\right] + n$$

Consider $16 \to$ as a negligible term

$$T(n) \leq 2\left[c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \log \left\lfloor \frac{n}{2} \right\rfloor\right] + n$$

$\leq \cancel{2} c \frac{n}{\cancel{2}} \log\left(\frac{n}{2}\right) + n$

$\leq cn \log n - cn \log 2 + n$

$\leq cn \log n - (c-1)n$ $\forall c \geq 1$ (for g)

$\leq cn \log n$

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

## Changing Variable Method

example: Consider the following recurrence

$T(n) = 2T(\sqrt{n}) + \log n$

solution: Suppose $m = \log_2 n \Rightarrow n = 2^m$

$n^{1/2} = 2^{m/2} \Rightarrow \sqrt{n} = 2^{m/2}$

Put the values, we get

$T(2^m) = 2T(2^{m/2}) + m$

Again consider

$S(m) = T(2^m)$

we have

$S(m) = 2S(m/2) + m$

We know that, this recurrence has the solution

$S(m) = O(m \log m)$

Substitute the values of m, we get

$T(n) = S(m) = O(\lg n \lg \lg n)$

example: Solve the recurrence :-

$T(n) = 2T(\sqrt{n}) + 1$ By making a use of changing variable method.

Suppose $m = \log n \Rightarrow 2^m = n \Rightarrow \sqrt{n} = 2^{m/2}$

Thus $T(2^m) = 2T(2^{m/2}) + 1$

$S(m) = T(2^m)$

$S(m) = 2S(m/2) + 1.$

we know the solution of above recurrence relation

$S(m) = O(\log m)$

# Assignment No. 1

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

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

(3) ✓ $T(n) = 2T(n/4) + n^{0.51}$

(4) $T(n) = 2T(n/4) + \sqrt{n}$

(5) $T(n) = 0.5\,T(n/2) + 1/n$

(6) $T(n) = 4T(n/2) + \log n$

(7) $T(n) = \sqrt{2}\,T(n/2) + \log n$

(8) $T(n) = 64T\left(\frac{n}{8}\right) - n^2\log n$

Solve all these Questions by Master's Theorem.

(9) $T(n) = T(9n/10) + n$

(10) $T(n) = 8T\left(\frac{n}{2}\right) + n^2$ (By Recurrence Tree method)

(11) $T(n) = 2T\left(\frac{n}{2}\right) + c \quad ; n > 1$
$\quad\quad = c \quad ; n = 1$ (By Recursion Tree method)

(12) $T(n) = 2T(n/2) + n \quad ; n > 1$
$\quad\quad = 1 \quad ; n = 1$ (By Recursion Tree method)

(13) $T(n) = 4T(n/2) + n$ by using iteration method.

(14) $T(n) = 8T(n/2) + n^2$ by using iteration method.

(15) $T(n) = 2T(\sqrt{n}) + \lg n$ by using substitution method.

(16) ✓ Solve the Recurrence

$$T(1) = 1$$
$$T(n) = 3T\left(\frac{n}{2}\right) + 2n^{1.5}$$

(17) Solve the recurrence
$$T(n) = T(n-1) + n.$$

(18) Solve the recurrence
$$T(n) = T\left(\frac{9n}{10}\right) + n.$$

... [illegible] Mole .

(19) Solve the recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$$

(20) Solve the recurrence

$$T(n) = 100T\left(\frac{n}{99}\right) + \log(n!)$$

Clue:- By stirling's approximation $\log n! = n\log n$.

(21) Consider the following recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

(22) Solve the following recurrence using recursion tree method $T(n) = 8T\left(\frac{n}{2}\right) + n^2$

(23) $T(n) = T(n/3) + n^{4/3}$. Solve this recurrence by iteration method.

ostituting for m, we get-

$T(n) = S(m) = O(\lg m) = O(\lg\lg n)$

Iteration Method

example. Consider the recurrence

$$T(n) = 3T\left(\lfloor n/4 \rfloor\right) + n.$$

$$T(n) = n + 3T\left(\lfloor n/4 \rfloor\right)$$

$$T(n) = n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\lfloor n/16 \rfloor\right)\right)$$

$$T(n) = n + 3\left(\lfloor n/4 \rfloor + 3\left(\lfloor n/16 \rfloor + 3T\left(\lfloor n/64 \rfloor\right)\right)\right)$$

$$T(n) = n + 3\left\lfloor \frac{n}{4} \right\rfloor + 9\left\lfloor n/16 \right\rfloor + 27T\left(\lfloor n/64 \rfloor\right)$$

$\because \frac{n}{4^k} = 1$

$n = 4^k$

$k = \log_4 n.$

$$T(n) = n + \frac{3n}{4} + \left(\frac{3}{4}\right)^2 n + \cdots\cdots + \left(\frac{3}{4}\right)^{k-1} n + 3^k T\left(\frac{n}{4^k}\right)$$

$$T(n) = n\left[1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \cdots\cdots \left(\frac{3}{4}\right)^{k-1}\right] + 3^{\log_4 n}T(1)$$

$$T(n) = n\sum_{i=0}^{k-1}\left(\frac{3}{4}\right)^i + 3^{\log_4 n}T(1)$$

$$T(n) \leq n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + 3^{\log_4 n}T(1)$$

$$\leq n\left[\frac{1}{1-3/4}\right] + n^{\log_4 3}\,T(1)$$

$$\leq 4n + n^{\log_4 3}\,T(1) \qquad \text{as } \log_4 3 > 1$$

$O(n)$ Ans

example   Solve this recurrence by iteration

$T(n) = T(n-1) + n^4$

Solution   $= [T(n-2) + (n-1)^4] + n^4$

(19)   $= T(n-3) + (n-2)^4 + (n-1)^4 + n^4$

- - - - - - - - - - - - - -

$= n^4 + (n-1)^4 + (n-2)^4 + \dots + 2^4 + 1^4 + T(0)$

$= \sum_{i=1}^{n} i^4 + T(0)$

$$\boxed{T(n) = \Theta(n^4)}$$

example   Solve the following recurrence relation:-

$T(n) = 2T(n/2) + 3n^2$

$T(1) = 11$

Solution   $T(n) = 2T\left(\frac{n}{2}\right) + 3n^2$

$= 3n^2 + 2T\left(\frac{n}{2}\right)$

$= 3n^2 + 2\left[3\left(\frac{n}{2}\right)^2 + 2T\left(\frac{n}{4}\right)\right]$

$= 3n^2 + 2\cdot 3\left(\frac{n}{2}\right)^2 + 4\left[3\cdot\left(\frac{n}{4}\right)^2 + 2T\left(\frac{n}{8}\right)\right]$

$= 3n^2 + \frac{3n^2}{2} + \cancel{4}\cdot 3\,\frac{n^2}{4^2} + 2^3 T\cdot\left(\frac{n}{2^3}\right)$

$T(n) = \frac{3n^2}{2^0} + \frac{3n^2}{2^1} + \frac{3n^2}{2^2} + \dots \frac{3n^2}{2^{k-1}} + 2^k T\left(\frac{n}{2^k}\right)$

The series terminates when $n/2^k = 1$   $n = 2^k$, $k = \log_2 n$

$T(n) = 3n^2\left[1 + \frac{1}{2} + \frac{1}{2^2} + \dots + \frac{1}{2^{k-1}}\right] + 2^k T(1)$

$= 3n^2\left[1 + \frac{1}{2} + \frac{1}{2^2} + \dots + \frac{1}{2^{k-1}}\right] + 2^{\log_2 n} T(1)$

$= 3n^2 \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i + n^{\log_2 2} T(1)$

$\leq 3n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + n\,T(1)$

$\leq 3n^2 \left[\frac{1}{1-\frac{1}{2}}\right] + 11n$ $\qquad S_\infty = \frac{1}{1-r}$

$\leq 3n^2 . 2 + 11n$

$\leq 6n^2 + 11n$

$T(n) \leq 6n^2 + 11n \Rightarrow T(n) = O(n^2)$

example $\quad T(n) = 1 \qquad$ for $n = 1$

$\qquad = 2T(n-1) \qquad$ for $n > 1$

Solve this recurrence relation.

$$\begin{aligned} T(n) &= 2T(n-1) \\ &= 2[2T(n-2)] = 4T(n-2) \\ &= 4(2T(n-3)) = 8T(n-3) \\ &= 2^3(T(n-3)) = 2^4 T(n-4) \end{aligned}$$

In general $T(n) = 2^i T(n-i)$ $\qquad n - i = 1$

$\qquad n = i+1$

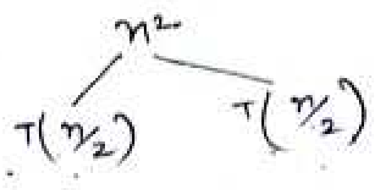
Put $i = n-1$ we get

$T(n) = 2^{(n-1)} T(1) = 2^{n-1}$ Ans.

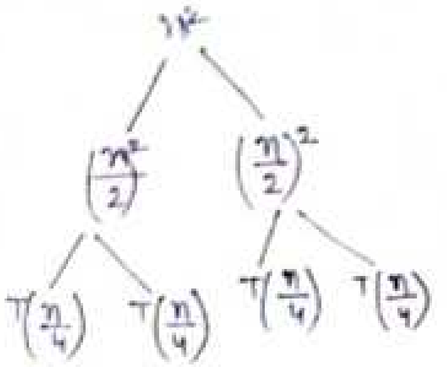
## Recursion Tree Method.

example Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$ we have to obtain the asymptotic bound using recursion tree method.

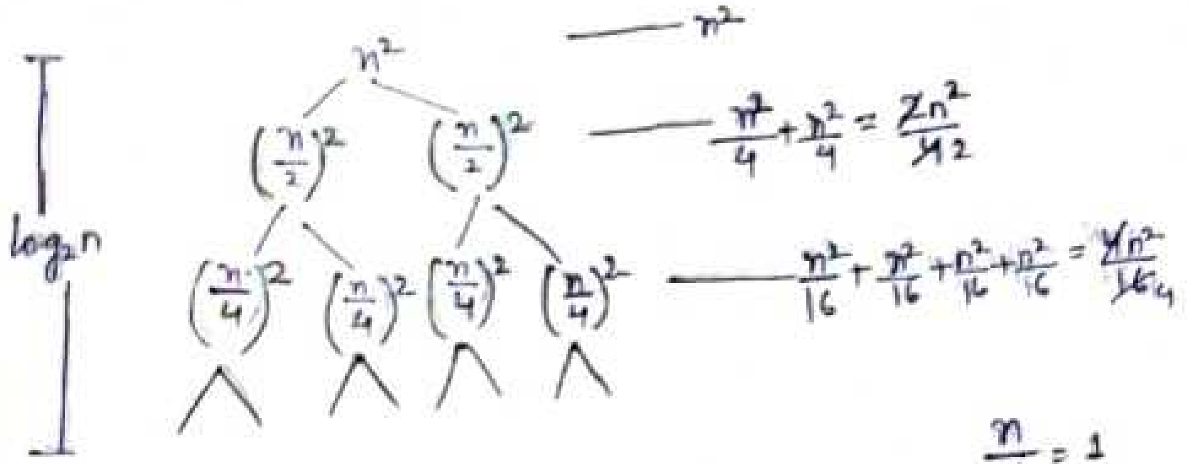
Solution $\quad T(n) = 2T\left(\frac{n}{2}\right) + n^2$



$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2.$



$n^2$
$\left(\frac{n^2}{2}\right)$
$\left(\frac{n}{2}\right)^2$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$



$n^2$
$n^2$
$\left(\frac{n}{2}\right)^2$
$\left(\frac{n}{2}\right)^2$
$\frac{n^2}{4}+\frac{n^2}{4}=\frac{\cancel{2}n^2}{\cancel{4}2}$
$\log_2 n$
$\left(\frac{n}{4}\right)^2$
$\left(\frac{n}{4}\right)^2$
$\left(\frac{n}{4}\right)^2$
$\left(\frac{n}{4}\right)^2$
$\frac{n^2}{16}+\frac{n^2}{16}+\frac{n^2}{16}+\frac{n^2}{16}=\frac{\cancel{4}n^2}{\cancel{16}4}$

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

$n=2^i$

$i=\log_2 n$

$T(n)= n^2+\frac{n^2}{2}+\frac{n^2}{4}+ \text{------} + \log_2 n \text{ times}$

$= n^2\left[1+\frac{1}{2}+\frac{1}{4}+ \text{-----} \log_2 n \text{ times}\right]$

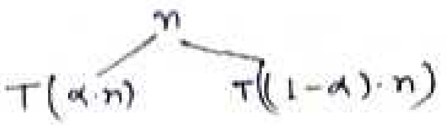
$\leq \sum_{i=0}^{\infty} n^2\left[\frac{1}{2}\right]^i$

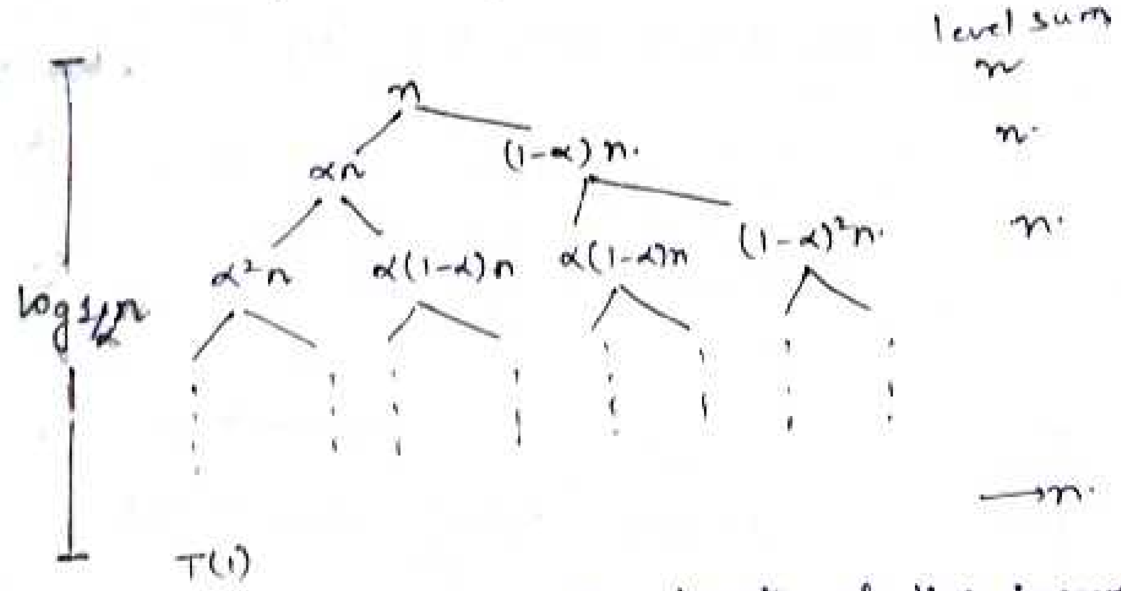
$\leq n^2 \sum_{i=0}^{\infty}\left[\frac{1}{2}\right]^i$

$\leq n^2\left[\frac{1}{1-1/2}\right] = O(n^2)$ Ans example Solve this recurrence by recursion tree method.

$$T(n) = T(\alpha \cdot n) + T(1-\alpha)\cdot n + n$$

Solution The recursion tree from the above recurrence is





we add the values across the levels of the recursion tree, we a get value of $n$ for every level. The Longest path from root to a leaf is.

$$\alpha^i n = 1$$

$$n = \left(\frac{1}{\alpha}\right)^i$$

$$i = \log_{1/\alpha} n.$$

Height of Tree is $n \cdot \log_{1/\alpha} n$.

number of nodes in each level is n.

So total number of nodes in last level = $O(n \log n)$

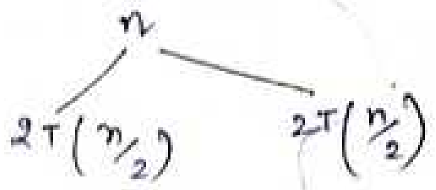
19 [ We have $n + n + n + n + \ldots \log_{1/2} n$ times.

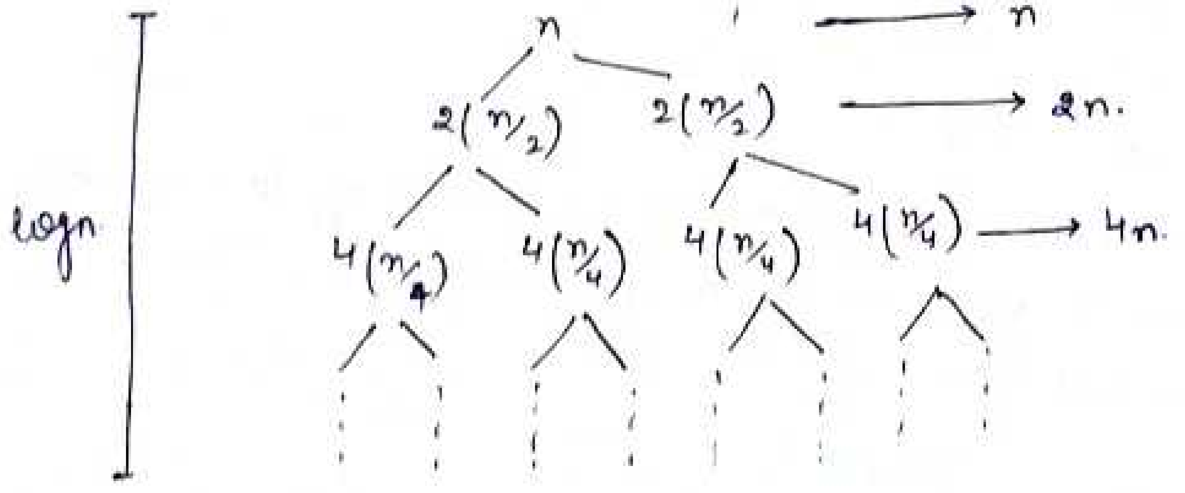
20 $= n[1+1+1+1+ \ldots \log_{1/2} n \text{ times}]$

$= n \times \log_{1/2} n$ Ans. ]

<u>example</u> $T(n) = 4T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n.$

Solve the above recurrence by recursion tree method.





$\frac{n}{2^i} = 1$ | $i = \log_2 n$

$i = \log_2 n$

$T(n) = n + n + n + \dots \log_{3/2} n$ times.

$= n\,[1 + 1 + 1 + \dots \log_{3/2} n \text{ times}]$

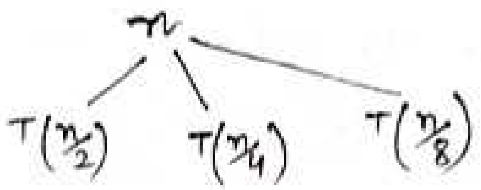
$T_n = \Theta(n \cdot \log_{3/2} n)$ Ans.

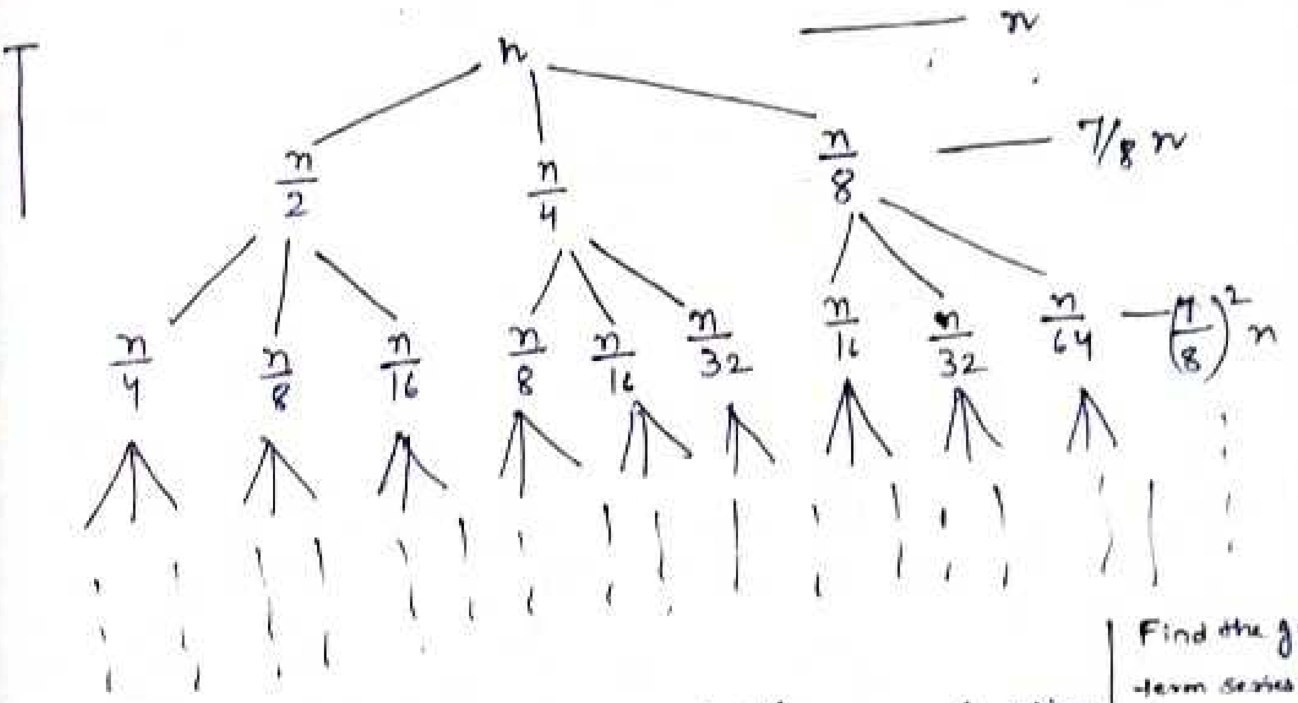
example :- Solve the following recursion

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n.$$ by

recursion tree method.

Solution





$n$ — $n$
$\frac{n}{2}$ $\frac{n}{4}$ $\frac{n}{8}$ — $\frac{7}{8}n$
$\frac{n}{4}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{64}$ — $\left(\frac{7}{8}\right)^2 n$

$$T(n) = n + \frac{7}{8}n + \left(\frac{7}{8}\right)^2 n + \dots \log_2 n \text{ times}$$

$$= n\left[1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \dots \log_2 n \text{ times}\right]$$

$$\le n \sum_{i=0}^{\infty} \left(\frac{7}{8}\right)^i$$

$$\le n\left[\frac{1}{1 - 7/8}\right]$$

$$\le 8n \qquad \boxed{T(n) = \Theta(n)}$$

Find the greater term series $\frac{n}{2^i}$,

$n = 2^i$

$i = \log_2 n$

## Master's Theor$\overset{e}{\wedge}$m

$$T(n) = aT\left(\frac{n}{b}\right) + \theta(n^k \log^p n)$$

$a \geq 1$ , $b > 1$ , $k \geq 0$ and $p$ is real number

1.) if $a > b^k$ , then $T(n) = \theta(n^{\log_b a})$

2.) if $a = b^k$

a) if $p > -1$ then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

b) if $p = -1$ then $T(n) = \theta(n^{\log_b a} \log\log n)$

c) if $p < -1$ then $T(n) = \theta(n^{\log_b a})$

3.) if $a < b^k$

a) if $p \geq 0$ then $T(n) = \theta(n^k \log^p n)$

b) if $p < 0$ then $T(n) = O(n^k)$

<u>NOTE :-</u> $(\log n)^2$ is completely different from $\log^2 n$.

$$(\log n)^2 \downarrow \log n * \log n \qquad \log^2 n \downarrow \log\log n.$$

<u>example 1:-</u> $T(n) = 3T\left(\frac{n}{2}\right) + n^2$

$a = 3$ $b = 2$ $p = 0$ $k = 2$

| | |
|---|---|
| $a$ | $b^k$ |
| $3$ | $2^2$ |
| $3$ | $< 4$ |

Here we apply the case (3)

$p = 0$ So we apply (a) case of (3)

$T(n) = \theta(n^2 \log^0 n) \rightarrow T(n) = \theta(n^2)$ Ans.

Example 2:- $T(n) = 4T(n/2) + n^2$

$a = 4 \quad b = 2 \quad p = 0 \quad k = 2$

$4 \quad 2^2$

$4 = 4$

Here we apply case (2)

$p > -1 \qquad T(n) = \theta(n^{\log_2 4} \log n)$

$= \theta(n^2 \log n)$ Ans.

example 3:- $T(n) = 2^n T(n/2) + n^n$

Generally a is constant but Here a is exponent, so the master's Theorm cannot be applied over here.

example 4:- $T(n) = 2T(n/2) + n/\log n$.

$T(n) = 2T(n/2) + n \cdot \log^{-1} n$.

$a = 2, \; b = 2, \; k = 1, \; p = -1$

$2 \quad 2^1$

$2 = 2$. Here we apply. the case (2)

$p = -1$.

$T(n) = \theta(n^{\log_2 2} \log\log n)$

$= \theta(n \log\log n)$

# Assignment No - 1

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

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

(3) $T(n) = 2T(n/4) + n^{0.51}$

(4) $T(n) = 2T(n/4) + \sqrt{n}$

(5) $T(n) = 0.5\,T(n/2) + 1/n$

(6) $T(n) = 4T(n/2) + \log n$

(7) $T(n) = \sqrt{2}\,T(n/2) + \log n$

(8) $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

Solve all these Questions by Master's Theorem.

(9) $T(n) = T(9n/10) + n$

(10) $T(n) = 8T\left(\frac{n}{2}\right) + n^2$ (By Recurrence Tree method)

(11) $T(n) = 2T\left(\frac{n}{2}\right) + c \;,\; n > 1$
$\quad = c \;,\; n = 1$ (By Recursion Tree method)

(12) $T(n) = 2T(n/2) + n \;;\; n > 1$
$\quad = 1 \;;\; n = 1$ (By Recursion Tree method)

(13) $T(n) = 4T(n/2) + n$ by using iteration method.

(14) $T(n) = 8T(n/2) + n^2$ by using iteration method.

(15) $T(n) = 2T(\sqrt{n}) + \lg n$ by using substitution method.

(16) Solve the recurrence

$$T(1) = 1$$
$$T(n) = 3T\left(\frac{n}{2}\right) + 2n^{1.5}$$

(17) Solve the recurrence
$$T(n) = T(n-1) + n.$$

(18) Solve the recurrence
$$T(n) = T\left(\frac{9n}{10}\right) + n.$$

(19) Solve the reccurence

$$T(n) = 4T(n/2) + n^2\sqrt{n}$$

(20) Solve the reccurence

$$T(n) = 100T(n/99) + \log(n!)$$

Clue :- By stirling's approximation $\log n! = n \log n$.

(21) Consider the following recurrence

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

Obtain the asymptotic bound using recursion tree method.

(22) Solve the following reccurence using recursion tree method $T(n) = 8T(n/2) + n^2$

(23) $T(n) = T(n/3) + n^{4/3}$. Solve this recurrence by iteration method.

// .

# UNIT-1.

## CHAPTER :- 5

## √MERGE SORT

Merge sort algorithm was invented by John von Neumann in 1945. It closely follow the Divide and conquer Paradigm:-

Conceptually, it works as follows-

1. Divide:- Divide the unsorted list into two sublists of about half the size.
2. conquer:- Sort each of the two sub lists recursively until we have list sizes of length 1
3. combine:- Merge the two sorted sub lists back into one sorted list.

Algorithm

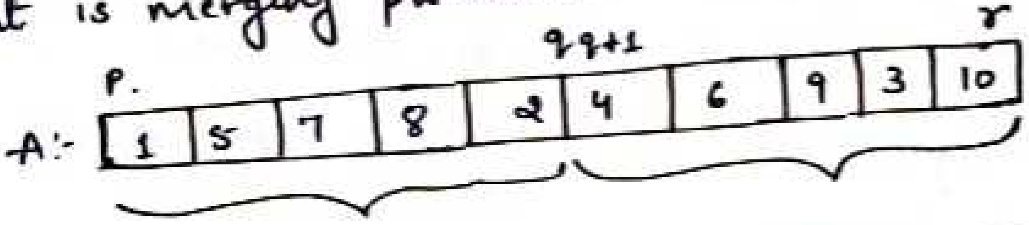
```
MERGE (A, P, q, r)
1. n1 ← q-p+1
2. n2 ← r-q
3. Let L[1 ..... n1+1] and R[1 to n2+1] be new arrays
4. for(i = 1 to n1)
5.         L[i] ← A[P+i-1]
6. for(j = 1 to n2)
7.         R[j] ← A[q+j]
8. L[n1 +1] ← ∞
9. R[n2 +1] ← ∞
```

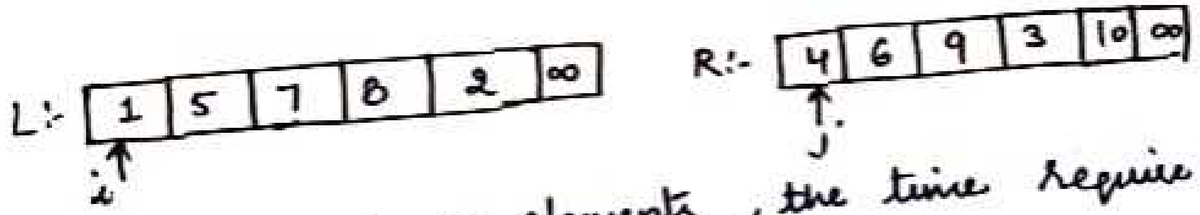
```
10. i = 1 , j = 1
11. for (k = p to r)
12.     if (L[i] ≤ R[j])
13.         A[k] ← L[i]
14.         i ← i+1
15.     else
16.         A[k] ← R[j]
17.         j ← j+1
18. end for
19. end.
```

→ poi
in
k→

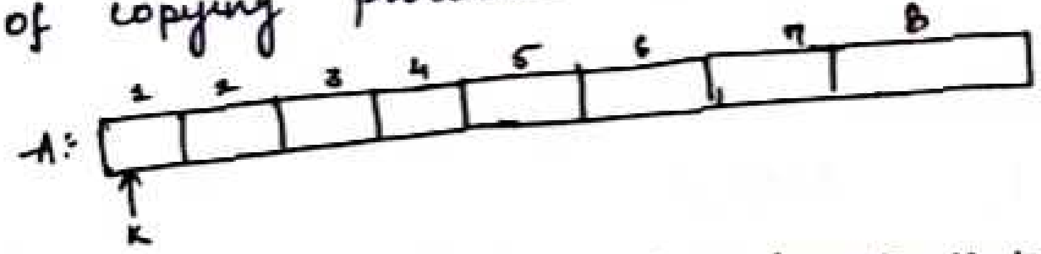
## Analysis of Merge :-

Merge Sort is better in compare to Insertion Sort as well as Quick Sort interms of time complexity. Heart of merge Sort is merging procedure.





Total of there are n elements, the time require to copy them from original unsorted list A to the newly created list L & R & the time taken to perform such kind of copying procedure is O(n)



i → pointer that indicates the elements that you are examining in the left list (L).

- j → pointer that indicates the elements that you are examining in the right list (R)

K → indicates the pointer that is used to fill the elements in the new array.

Compare i & j pointers pointing elements & write down that element in the array A which is lesser.
If the element pointed by i pointer is lesser write down that element in the array A after that i pointer moves forward
Similarly for j pointer.

To write down 1 element we have to do one comparision
So If there are n elements then we have to do

$$\underbrace{(n}_{\text{comparisions}} + \overset{}{n}) = O(n)$$

$n \longrightarrow$ copies

So, The total time complexity of Merging procedure is $O(n)$

```
Merge-Sort (A, p, r) —— T(n)
{  if (p<r)            —— O(1)
      q = ⌊(p+r)/2⌋        —— O(1)
      Merge-Sort (A, p, q) —— T(n/2)
      Merge-Sort (A, q+1, r) —— T(n/2)
      Merge (A, p, q, r)   —— O(n)
}
```

## Analysis of Merge Sort

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

By applying master's Theorm

$\theta(n \log n)$

$a = 2 \quad b = 2 \quad k = 1$ [illegible]

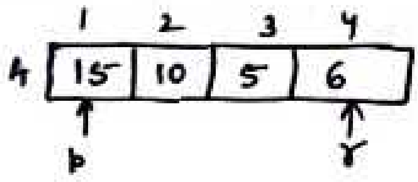
$2 = 2^1 \rightarrow a = b^k$

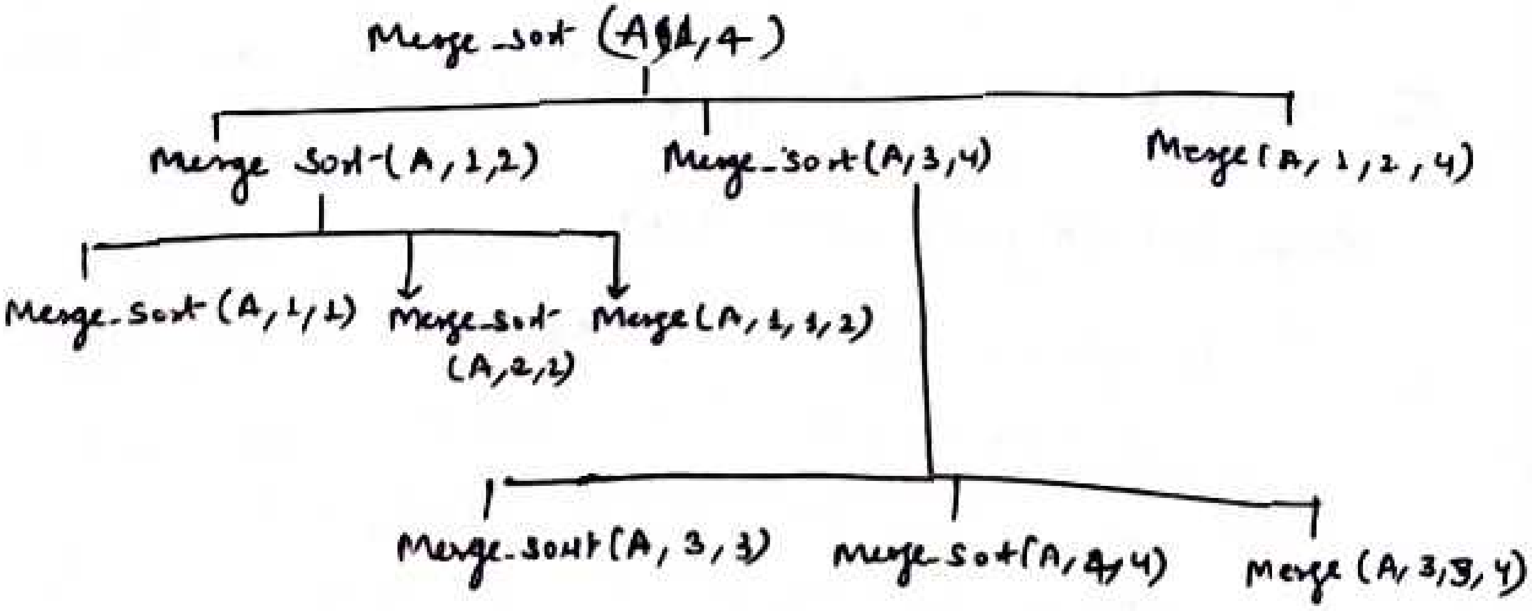
$p = 0 \rightarrow p > -1$

$T(n) = \theta(n^{\log_2 2} \log^{0+1} n)$

$= \theta(n \log n)$ ✓

example :-
```
Merge_Sort (A, 1, 4)
{ if (1<4)
    q = ⌊(1+4)/5⌋ = ⌊2.5⌋ = 2.
    Merge Sort (A, 1, 2)
    Merge Sort (A, 3, 4)
    Merge (A, 1, 2, 4)
}
```





Merge-Sort (A, 1, 2)

```
{
    if (1 < 2) True.
        q = ⌊(1+2)/2⌋ = ⌊3/2⌋ = 1.
        Merge-sort (A, 1, 1)
        Merge-sort (A, 2, 2)
        Merge (A, 1, 1, 2)
}
```

$A\ \begin{array}{|c|c|} \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} \\ \hline 15 & 10 \\ \hline \end{array}$

```
Merge-sort (A, 1, 1)
    {
        if (1 < 1) false
    }
```

$A\ \begin{array}{|c|} \multicolumn{1}{c}{1} \\ \hline 15 \\ \hline \end{array}$

```
Merge-sort (A, 2, 2)
    {
        if (2 < 2) false
    }
```

$A\ \begin{array}{|c|} \multicolumn{1}{c}{2} \\ \hline 10 \\ \hline \end{array}$

Merge $(A, \underset{p}{1}, \underset{q}{1}, \underset{r}{2})$

1. $n_1 = 1 - 1 + 1$

   $n_1 = 1$

2. $n_2 = 2 - 1 = 1$

3. $L[1, 2] \quad R[1, 2]$

   $L\ \begin{array}{|c|c|} \hline 15 & \\ \hline \end{array} \qquad \begin{array}{|c|c|} \hline 10 & \\ \hline \end{array}$

5. $i = 1$ to $1$

   $L[1] \leftarrow A[1 + 1 - 1]$

   $L[1] \leftarrow A[1]$

6. $j = 1$ to $1$

   $R[1] \leftarrow A[1 + 1]$

```
8.   L[1+1] ⇒ L[2] = ∞
9.   R[1+1] ⇒ R[2] = ∞

 L |15|∞|          R |10|∞|

10. i=1 , j=1
11. {for( k=1 to 2)
         if (L[1] ≤ R[1])      (False)
              15 ≤ 10
         else
              A[1] ← R[1]
              A[1] ← 10
              j ← j+1 = 2
     for(k=2 to 2)
         if (L[1] ≤ R[2])
              15 ≤ 100          (True)
               15 ≤ ∞
            A[2] ← L[1]
            A[2] ← 15
            i = 2.
```

```
                                      A |10|15|
```

```
Merge-Sort(A,3,4)
{
 if (3<4) False
      q = ⌊(3+4)/2⌋ = ⌊7.5/2⌋ = 3
    Merge-Sort(A,3,3)
     Merge-sort(A,4,4)
     Merge (A,3,3,4)
 }
Merge-Sort(A,3,3)
 { if(3<3) False)
 }
```

```
Merge-Sort(A,4,4)
  {
   if A(4<4)
               False)
  }
Merge(A,3,3,4)
 {
1) n1 ← 3-3+1
   n1 ← 1
2) n2 ← 4-3
   n2 ← 1
```

```
3). L [1,2]     R[1,2]
4)  for i=1 to 1
      L[1] ← A[3+1 -1]
      L[1] ← A[3]
      L[1] ← 5
5)  for j=1 to 1
      R[1] ← A[3+1]
      R[1] ← A[4]
      R[1] ← 6
6)  L[1+1] ← ∞
    R[1 +1] ← ∞
7)  i=1
8)  j=1
9)  for k= 3 to 4
        if L[1] ≤ R[1]
            5 ≤ 6    True.
        A[3] ← L[1]
            i=2
    for k=4 to 4
        if L[2] ≤ R[1]
            ∞ ≤ 6   (False)
        A[4] ← R[1]
        A[4] ← 6
        j=2
```

L | 5 | |

R | 6 | |

L= | 5 | ∞ |

R= | 6 | ∞ |

A | 10 | 15 | 5 | 6 |

Sorted . Array | 10 | 15 | 5 | 6 |

//

# CHAPTER-6 (UNIT-1)

## QUICK SORT

The basic version of quick sort algorithm was invented by C.A.R Hoare in 1960 and formally introduced quick sort in 1962. It is used on the principle of divide and conquer. Quick sort is an algorithm of choice and it is also in-place sorting algorithm

### Advantages of Quick Sort

1. It is in-place since it uses only a small auxiliary stack.
2. It requires only n log(n) times to sort n items.
3. It has an extremely short inner loop.

### Disadvantages of Quick Sort

1. It is recursive. Especially if recursion is it not available, the implementation is extremely difficult.
2. It requires quadratic time in worst case i.e $n^2$

### Working of Quick Sort

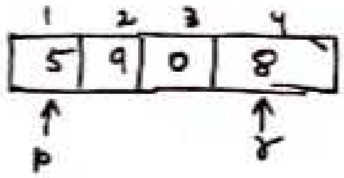
1. Quick sort works by partitioning a given array A[p...r] into two non empty sub arrays A[p..q] and A[q+1...r] such that every key in A[p..q] is less than or equal to every key in A[q+1...r]

2. The two sub arrays are sorted by recursive calls.

Algorithm

```
QUICK_SORT(A, p, r)
1. If p < r then
2.         q ← PARTITION (A, p, r)
3.         QUICK_SORT(A, p, q-1)
4.         QUICK_SORT(A, q+1, r)

PARTITION (A, p, r)
1. x = A[r]
2. i = p-1
3. for (j = p to r-1)
   {
4.     if (A[j] ≤ x)
5.     {  i = i+1
6.        exchange A[i] with A[j]
   }  }
7.     exchange A[i+1] with A[r]
8.  }  return i+1
   }
```

example :-



if 1<4

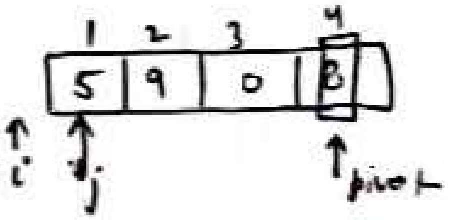
q ← PARTITION(A, 1, 4)

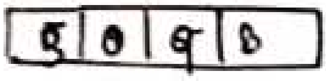
PARTITION (A, 1, 4)

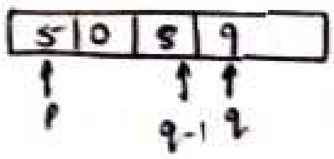
1. x = A[4] = 8

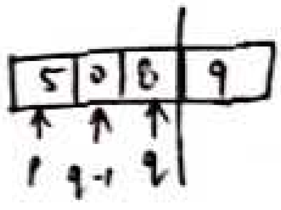
2. i = 0

3. for (j = 1 to 3)

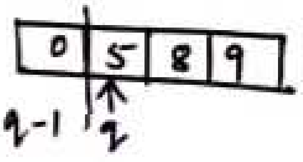
```
1:  if A[1] ≤ 8
        5 ≤ 8
        i = 1
        A[1] ⟷ A[1]
    for j = 2 to 3
        if A[2] ≤ 8
            9 ≤ 8   (False)
    for j = 3 to 3
        if A[3] ≤ 8
            0 ≤ 8 (True)
            i = 2
            A[2] ⟷ A[3]
    A[3] ⟷ A[4]
Return  q ← 4

    QUICK-SORT(A, 1, 3)
    if 1 < 3
        q ← PARTITION(A, 1, 3)

    PARTITION(A, 1, 3)
    1. x = A[3]      x = 8
    2. i = 0
    3. for (j = 1 to 2)
         if (A[1] ≤ 8)
             5 ≤ 8  (False)
         {  i = 1
            A[1] ⟷ A[1]
         }
```








```
for ( j = 1 to 2 )
    if A[1] ≤ 8
        0 ≤ 8 (True)
        i = 2
        A[2] ⟷ A[1]

exchange A[3] ⟷ A[3]
Return q = 3.

QUICK-SORT( A, 1, 2 )
If ( 1 < 2 )
    q ← PARTITION( A, 1, 2 )

PARTITION( A, 1, 2 )
1. x ← A[2] = 0
2. i = 1 - 1 = 0
3. for j = 1 to 1
       do if A[1] ≤ 0
           5 ≤ 0   (False)
   exchange A[1] ⟷ A[2]
   return i = 2

   q ← 2

   QUICK-SORT( A, 1, 1 )
   If 1 < 1   ( False )
```



## Analysis of Quick Sort

### Time complexity of Partition Algorithm

By the end of 1 time partition algorithm 1 element is sorted and placed at its proper place.

Here j loop is running from first element to one less than the last element

So the total time taken by partition algorithm is

$O(n-1)$ i.e $O(n)$

### Time Complexity of Quick Sort

### Best case time Complexity

Time taken to sort $n$ elements is $T(n)$.

In Best case Partition algorithm is going to given mid point as pivot element.

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

By applying Master's theorm

$$= \Theta(n \log n)$$

```
QUICK-SORT (A, p, r) → T(n)
  if(p<r)
    { q = QUICKSORT (A, p, r) – Θ(n
      QUICKSORT (A, p, q-1) – T(n/2
      QUICKSORT(A, q+1, r) – T(n/2
    }
```

## Worst case Time Complexity

Here 0 element on the one side of partition and all the other $(n-1)$ elements are on other side and 1 element already in its correct position.

$$\underbrace{\boxed{2}\boxed{8}\boxed{6}\boxed{1}}_{(n-1)\text{ elements}}\boxed{10}\;\underset{\text{el}}{0}$$

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

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

$$\leq T(n-2) + c(n-1) + cn$$

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

$$\leq T(n-4) + c(n-3) + c(n-2) + c(n-1) + cn$$

$$\leq c1 + c_2 + c_3 + \cdots + c(n-3) + c(n-2) + c(n-1) + c(n)$$

$$\leq c[1+2+3+ \cdots + (n-3) + (n-2) + (n-1) + n]$$

$$\leq c\left[\frac{n(n+1)}{2}\right]$$

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

So total running time of Quicksort algorithm in the worst case is $O(n^2)$

**Imp Note**

If all the elements of Quicksort algorithm are arranged in ascending order.

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

$$\leq T(n-2) + c(n-1) + cn$$

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

$$\leq c1 + c2 + c3 + c*4 + \cdots c(n-2) + c(n-1) + cn$$

$$\leq c[1 + 2 + 3 + \cdots (n-2) + (n-1) + n]$$

$$\leq c\left[\frac{n(n+1)}{2}\right]$$

$$= O(n^2)$$

Similarly If all elements are arranged in descending the time complexity of QuickSort algorithm remain $O(n^2)$

## Randomized Quick Sort

In the randomized version of Quick sort we impose a distribution on input. In this version we choose a "random key" for the pivot.

Assume that procedure Random (a, b) returns a random integer in the range [a, b); there are b-a+1 integers in the range and procedure is equally likely to return one of them.

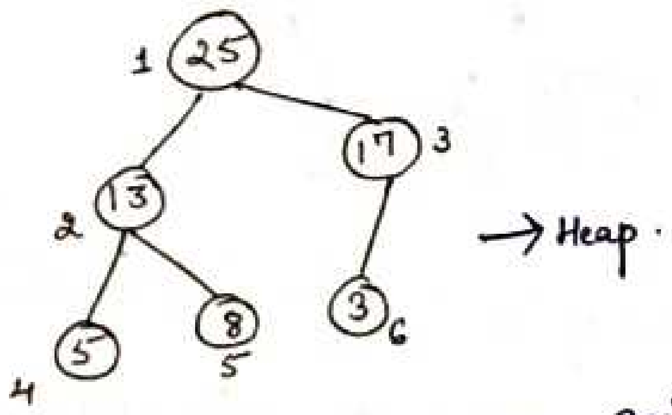
```
RANDOMIZED-PARTITION (A, p, r)
1.  i ← RANDOM(p, r)
2.  exchange A[p] ↔ A[i]
3.  return PARTITION (A, p, r)
```

```
RANDOMIZED-QUICKSORT (A, p, r)
1. if p < r
2.    then q ← RANDOMIZED-PARTITION(A, p, r)
3.          RANDOMIZED-QUICKSORT(A, p, q)
4.          RANDOMIZED-QUICKSORT(A, q+1, r)
```

```
PARTITION (A, p, r)
1. x ← A[r]
2. i ← p-1
3. for j ← p to r-1
4.      do if A[j] ≤ x
5.            then i ← i+1
6.                 exchange A[i] ↔ A[j]
7. exchange A[i+1] ↔ A[r]
8. return i+1
```

# CHAPTER-6
# HEAP SORT (UNIT-1)

**Binary Heap :-** The binary heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completed filled on all levels except possibly lowest



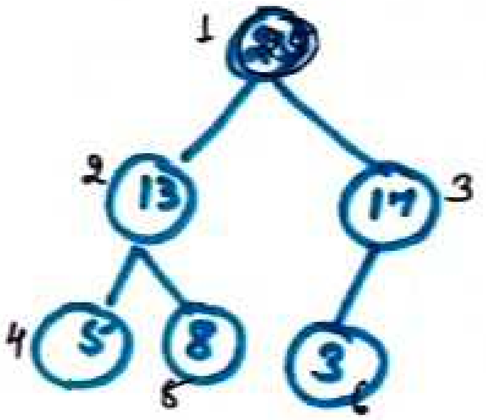
We represents heaps in level order, going from left to right. The array corresponds to the heap above is

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A: | 25 | 13 | 17 | 5 | 8 | 3 |

$$\boxed{\text{Heapsize [A] = length [A] = number of elements}}$$

The root of the tree A[1] and given index i of a node the indices of its parent, left child and right child can be computed :-

```
PARENT(i)
    return floor(i/2)
LEFT(i)
    return 2i
RIGHT(i)
    return 2i+1
```



The index of 25 is 1

To find the index of the left child, we calculate

2 * 1 = 2 i.e 13 (Key value of node)

To find the index of the right child, we calculate

2 * 1 + 1 = 3 i.e 17 (key value of node)

Heap Property

In a MAX-HEAP, for every node i other than the root, the value of a node is greater than or equal to the value of its parent

A[PARENT(i)] ≥ A[i]

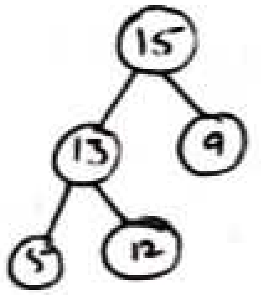
## Heap-Insert

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes the key of new element as the input into max-heap A. The procedure first expands the max-heap by adding to the tree a new whose key is $-\infty$. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

```
    HEAP-INCREASE_KEY(A, i, key)
1.  If key < A[i]
2.          then error " new key is smaller than current key".
3.  A[i] ← key.
4.  while i > 1 and (A[PARENT(i)] < A[i])
5.       do exchange A[i] ⟷ A[PARENT(i)]
6.          i ← PARENT(i)
```

```
    MAX_HEAP_INSERT(A, key)
1   heap-size[A] ← heap-size[A]+1
2.  A[heap-size[A]] ← -∞
3.  HEAP_INCREASE-KEY(A, heap-size[A], key)
```

example :-



|  |  |  |  |  |
|---|---|---|---|---|
| 15 | 13 | 9 | 5 | 12 |

(MAX-HEAP)

INSERT key = 10.

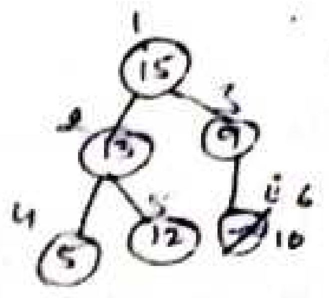
MAX_Heap-Insert(A, 10)

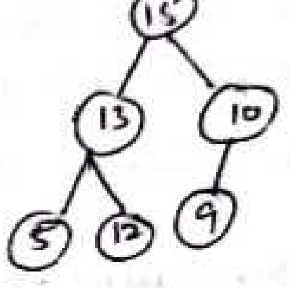
heap-size [A] ← 5+1 = 6

A[6] ← $-\infty$

HEAP_INCREASE_KEY ( A, 6, 10)

```
HEAP.INCREASE.KEY (A, 6, 10)
                      i  key

If 10 < -∞ (False)
A[6] ← 10
while 6 > 1 and A[3] < 10 (True)
        A[6] ⟷ A[3]
        i ← PARENT(i)
        i ← 3
while 3 > 1 and A[15 < 9] (False)
```

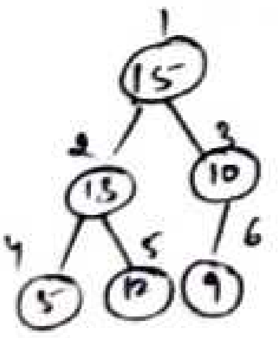




## HEAP.DELETE

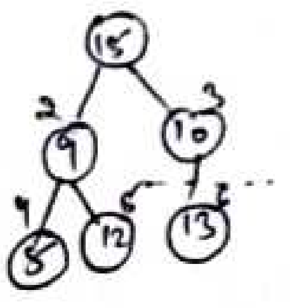
HEAP_DELETE (A,i) is the procedure, which deletes the item in node i from heap A. HEAP DELETE runs in O(lg n) time for an n element max-heap

```
HEAP_DELETE (A,i)
1.  A[i] ← A[heap-size[A]]
2.  heap-size[A] ← heap-size[A] - 1
3.  MAX-HEAPIFY (A,i)
```

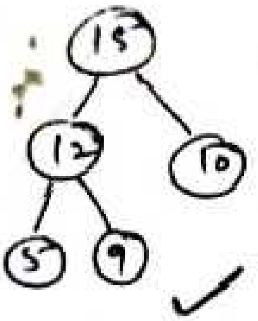


DELETE key = 2 (i)

```
A[2] ← A[6]
heap-size = 5
MAX-HEAPIFY (A,2)
1. l = 4        r = 5
2. 4 ≤ 6 and 5 > 9 (False)
        largest ← 2
3. 5 ≤ 6 and 12 > 9 (True)
        largest ← 5
4. 5 ≠ 2
        A[5] ⟷ A[2]
```



and 12 > 9 (True)



Final Answer.

MAX-HEAPIFY(A,5)

MAX-HEAPIFY(A,5)

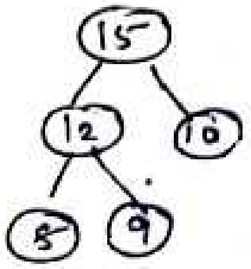
$l = 10$
$r = 11$ } (false)

HEAP-EXTRACT-MAX →

This operation removes and returns element having largest key value from the set.

```
HEAP-EXTRACT-MAX (A)
1. If heap-size [A] < 1
        then error " heap Underflow"
2.  max ← A[1]
3.  A[1] ↔ A[heap-size [A]]
4.  heap-size[A] ← heap-size[A]-1
5.  MAX-HEAPIFY (A,1)
6.  return max.
```

eg:- HEAP-EXTRACT-MAX (A)



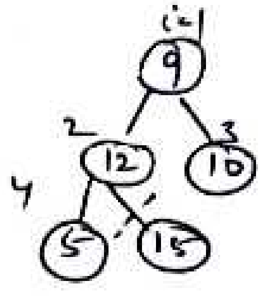
1. $5 < 1$ (False)

max ← A[1] ⇒ Max ← 15

A[1] ↔ A[5]

Heap size[A] ← 4

MAX-HEAPIFY (A,1)



1. $l = 2$
2. $r = 3$
3. $2 \le 5$ and $12 > 9$ (True)

heap is stored at

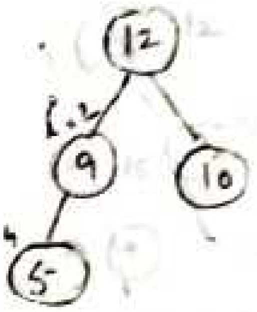
largest ← 2

3 ≤ 5 and 10 > 12 ( False)

2 ≠ 1

    A[1] ⟷ A[2]

    MAX-HEAPIFY( A, 2)



MAX-HEAPIFY (A, 2)

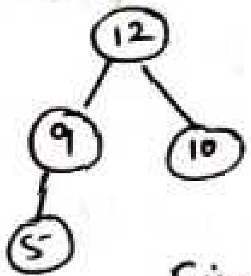
l = 4

r = 5

4 ≤ 5 and 5 > 9 (False)
    largest = 2

5 ≤ 5 and (False)

2 ≠ 2 (false)

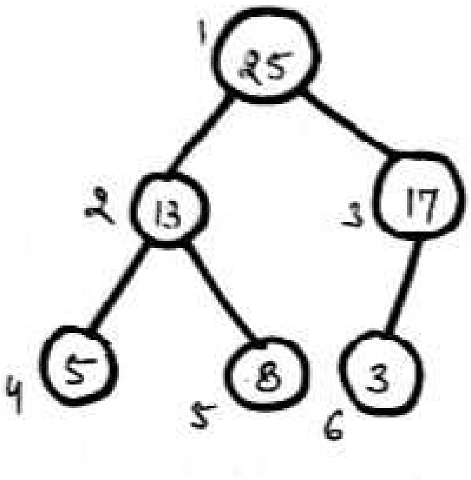


Final Tree

After Extraction of Maximum Number 15

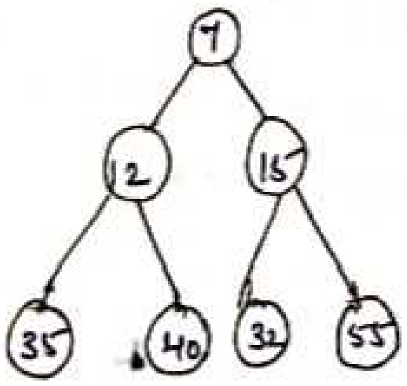
Thus, the largest element in a heap is stored at the root.

Following tree is an example of MAX-HEAP



Min-Heap is organised in the opposite way i.e the min Heap property is that for every node i other than the root is

$$A[\text{PARENT}(i)] \leq A[i]$$



Thus, the smallest element in a heap is stored at the root.

Height of a Heap

We define the height of a node in a tree to be a number of edges on the longest simple downward path from a node to a leaf.

Height of a tree

The number of edges on a simple downward path from a root to a leaf.

Note that the height of a tree with n node is $\lfloor \lg n \rfloor$ which is $\theta(\lg n)$. This implies that an n-element heap has height $\lfloor \lg n \rfloor$

Proof:- In order to show this let the height of n-element heap be h. From the bounds obtained on maximum & minimum number of elements in a heap, we get

$$2^h \le n \le 2^{h+1} - 1$$

Where n is the number of elements in a heap

$$2^h \le n \le 2^{h+1}$$

Take log on both side with base 2

$$h \le \lg n \le h+1$$

It follows that $h = \lfloor \lg n \rfloor$

Clearly, a heap of height h has the minimum number of elements when it has just one node at the lowest level. The levels above the lowest level form a complete binary tree of h-1 and $2^h - 1$ nodes.

Hence, the minimum number of nodes possible in a heap of height h is $2^h$.

Clearly, a heap of height $h$ has the maximum number of elements when its lowest level is completely filled. In this case the heap is a complete binary tree of height $h$ and hence has $2^{h+1}-1$ nodes.

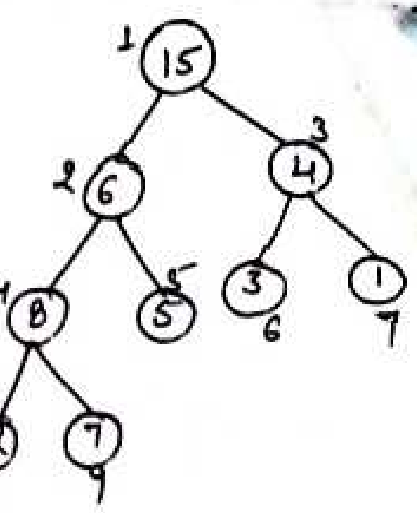
## Outline of Heapify Procedure

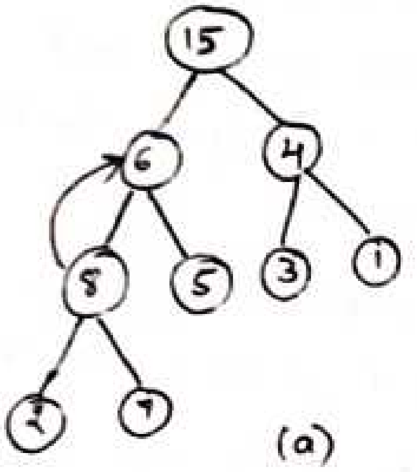
Heapify picks the largest child key and compares it to the parent key. If parent key is larger then heapify quits. Otherwise it swaps the parent key with the largest child key. So that the parent now becomes larger than its children.

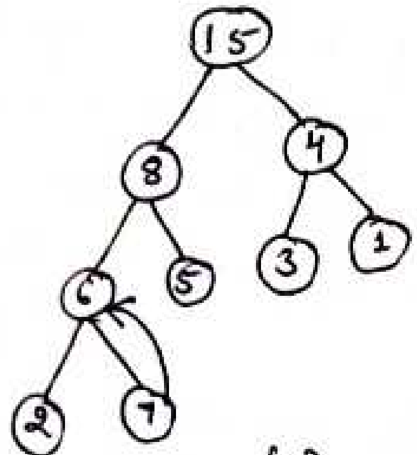
It is important to note that swap may destroy the heap property of the subtree rooted at the largest child node. If this is the case, heapify calls itself again using largest child node as the new root.

```
MAX HEAPIFY(A, i)
1.  l ← left[i]
2.  r ← right[i]
3.  if l ≤ heap-size[A] and A[l] > A[i]
4.        then largest ← l
5.        else largest ← i
6.  if r ≤ heap-size[A] and A[r] > A[largest]
7.        then largest ← r
8.  if largest ≠ i
9.        then exchange A[i] ↔ A[largest]
10.            MAX-HEAPIFY(A, largest)
```

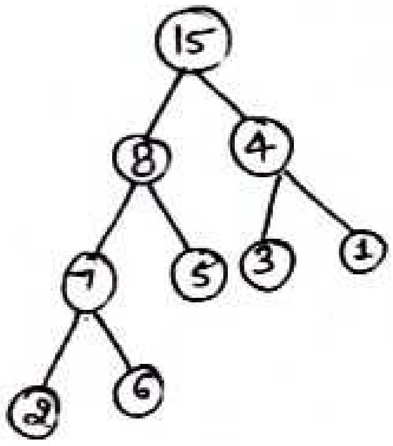
In this complete binary tree, the subtrees of 6 are not max-heap.
So we call MAX-HEAPIFY(A,2)





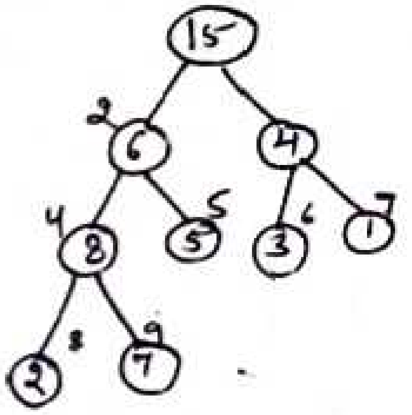


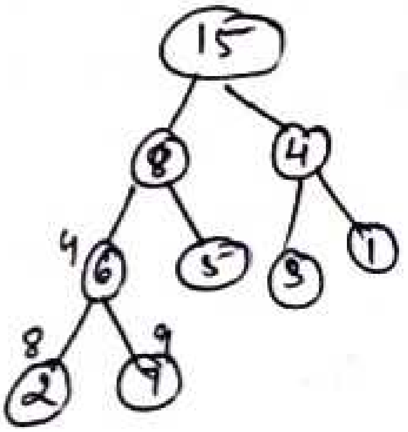
(a) ⇒ (b)



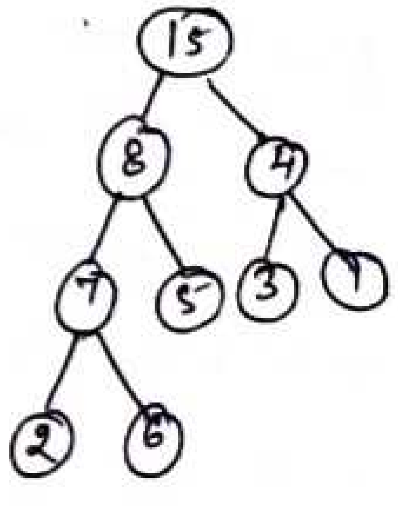
(c)

```
IAX-HEAPIFY (A,2)
l = 4
2. r = 5
3. 4 ≤ 9   and  8 > 6  (True)
        largest ← 4
5   5 ≤ 9   and  A[r] > A[4]
                      5 > 8  (False)
     if 4 ≠ 2
          exchange A[2] ⟷ A[4]
          MAX-HEAPIFY(A, 4)
```





```
MAX-HEAPIFY(A,4)
l = 8
r = 9
8 ≤ 9  and  2 > 6    (False)
     largest ← 4
9 ≤ 9  and  7 > 6  (True)
     largest ← 9
if 9 ≠ 4
     exchange A[4] ⟷ A[9]
```



## Building a Heap

We can use the procedure "Heapify" in a boti up fashion to convert an array A[1....n] into a heap

```
BUILD-MAX-HEAP(A)
1.  Heap-Size(A) ← length(A)
2.  For i ← floor (length[A]/2) down to 1 do.
3.          MAX-HEAPIFY( A,i)
```

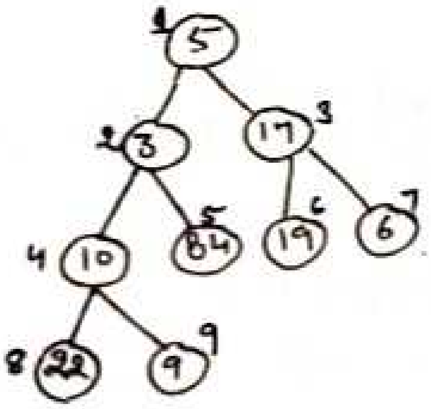
## Heapsort Algorithm.

The heap sort algorithm starts by using procedure MAX-BUILD-HEAP to build a heap on the input array A[1....n]. Since the maximum element of the array stored at the root A[1], it can be put into correct final position by exchanging it with A[n] (last element in A). If we now discard node n from the heap then the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property. Heap sort is an in-place sorting algorithm like an insertion sort.

```
HEAP-SORT(A)
1. BUILD-MAX-HEAP(A)
2. for i ← length[A] down to 2
3.     do exchange A[1] ←→ A[i]
4.         heap-size[A] ← heap-size[A] - 1
5.         MAX-HEAPIFY (A,1)
```

example :- Illustrate the operation of BUILD-MAX-HEAP on array

A = {5, 3, 17, 10, 84, 19, 6, 22, 9}



Heap size[A] = length[A] = 9.

for i ← ⌊9/2⌋ down to 1 do

→ for i ← 4 down to 1 do,

Max-Heapify (A, 4)

MAX-HEAPIFY (A,4)

l ← 8

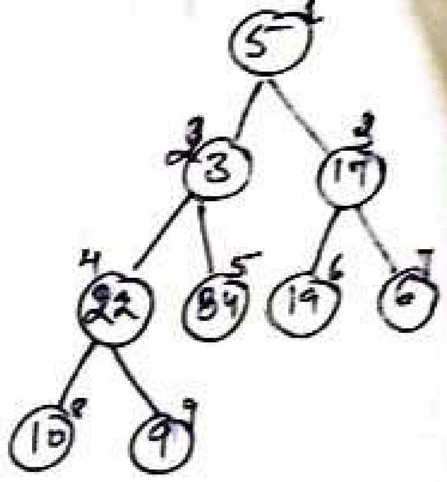
r ← 9

if 8 ≤ 9 and A[8] > A[4]
22 > 10 (True)

largest ← 8

if 9 ≤ 9 and A[9] > A[8]
9 > 22 ( False)

8 ≠ 4
exchange A[4] ⟷ A[8]

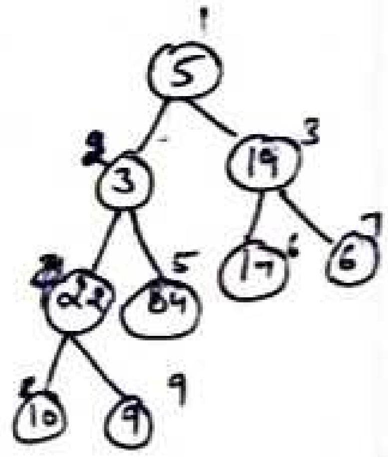


→ for i = 3 down to 1
MAX- HEAPIFY (A,3)

Max-Heapify (A,3)

1. l = 6
2. r = 7
3. 6 ≤ 9 and A[6] > A[3]
19 > 17 (True)
largest ← 6
4. 7 ≤ 9 and A[7] > A[6]
6 > 19 (False)

5. 6 ≠ 3
A[3] ⟷ A[6]



→ for i = 2 down to 1
MAX-HEAPIFY ( A,2)

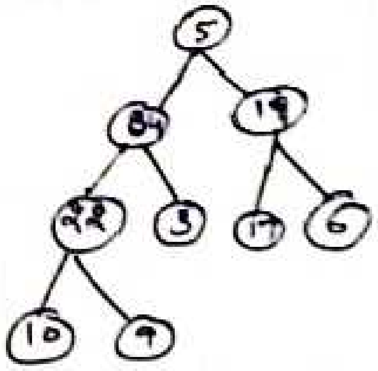
MAX- HEAPIFY (A,2)

1. l = 4
2. r = 5
3. 4 ≤ 9 and 22 > 3 (True)
largest ← 4
4. 5 ≤ 9 and 84 > 22 (True)

largest ← 5

$5 \neq 2$

A[2] ⟷ A[5]



→ for i = 1 down to 1.

MAX-HEAPIFY(A, 1)

MAX-HEAPIFY(A, 1)

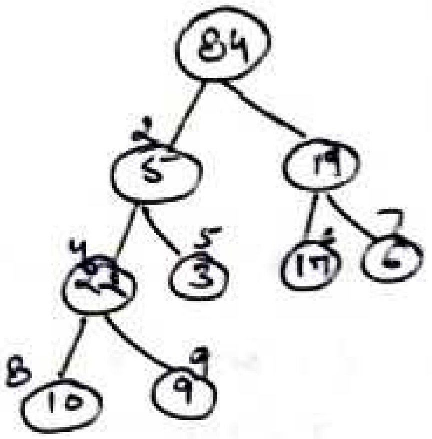
1. $l = 2$
2. $r = 3$
4. $2 \leq 9$ and $84 > 5$ (True)

   largest ← 2.

5. $3 \leq 9$ and $19 > 84$ (False)
6. $1 \neq 2$

   A[1] ⟷ A[2]

   MAX-HEAPIFY(A, 2)



MAX-HEAPIFY(A, 2)

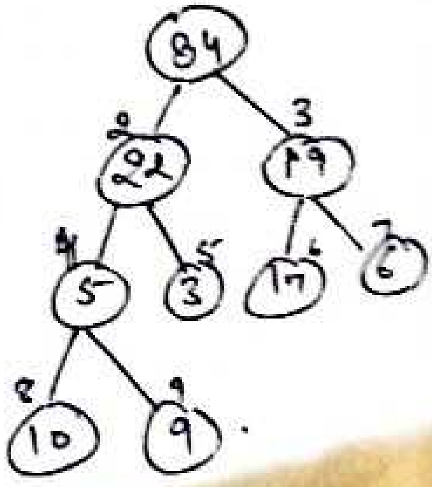
1. $l = 4$
2. $r = 5$
3. $4 \leq 9$ and $22 > 5$ (True)

   largest ← 4

4. $5 \leq 9$ and $3 > 22$ (False)
5. $2 \neq 4$

   A[2] ⟷ A[4]

   MAX-HEAPIFY(A, 4)



MAX-HEAPIFY (A,4)

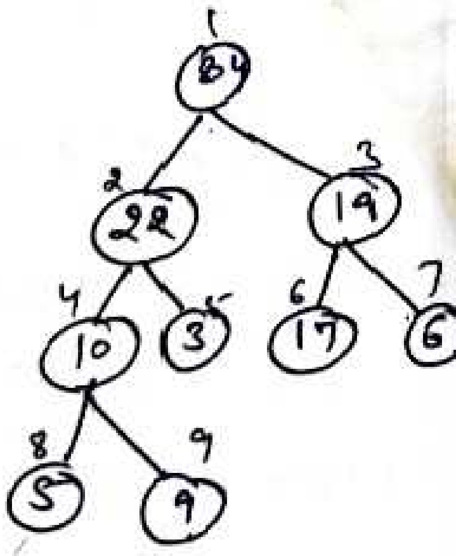
1. $l = 8$
2. $r = 9$
3. $8 \leq 9$ and $10 > 5$ (True)

   largest $\leftarrow 8$

4. $9 \leq 9$ and $9 > 10$ (False)
5. $8 \neq 4$

   $A[4] \leftrightarrow A[8]$

   MAX-HEAPIFY (A,8)



MAX-HEAPIFY (A,8)

$l =$ ~~[illegible]~~ 16

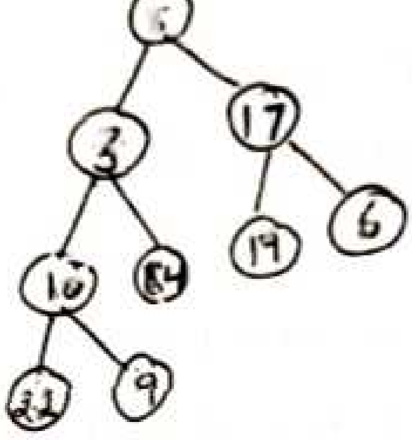
$r =$ ~~[illegible]~~ 17

~~[illegible]~~ 17 $\leq 9$ (False).

} False.

mple 9:- Illustrate the operation of HEAP-SORT on the array A = {5, 3, 17, 10, 84, 19, 6, 22, 9}

Solution

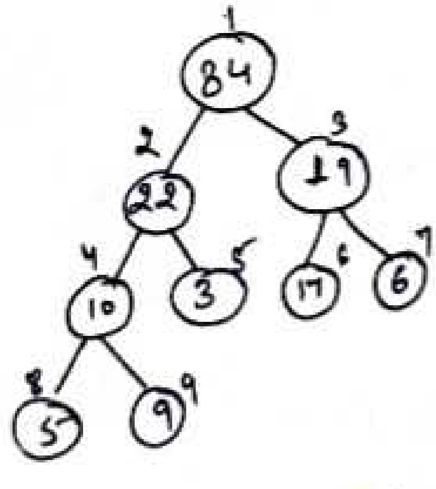


HEAP-SORT (A)

1. BUILD MAX-HEAP(A)

heapsize [A] ← 9

After a BUILD-MAX-HEAP(A).

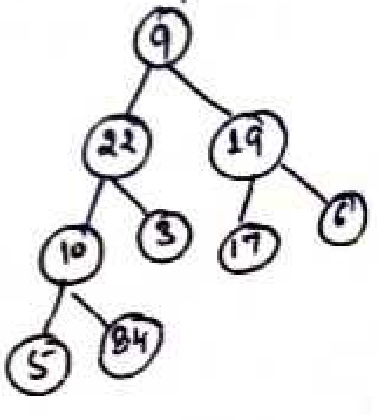


for i ← 9 down to 2

exchange A[1] ↔ A[9]

heapsize[A] ← 8

MAX-HEAPIFY (A, 1)



– MAX-HEAPIFY (A, 1)

1. l = 2   2. r = 3

3. If 2 ≤ 8 and 22 > 9 (True)

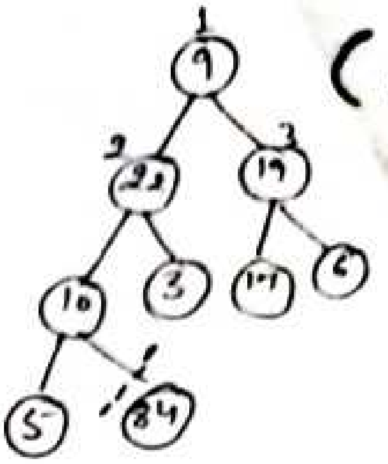
largest ← 2

4. If 3 ≤ 8 and 19 > 22 (False)

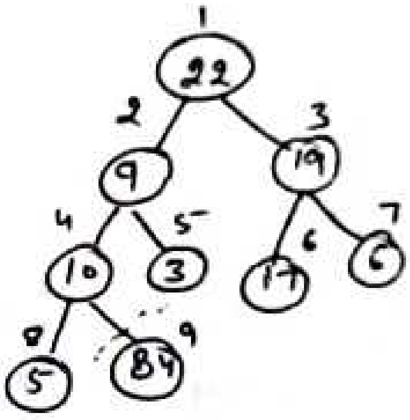
5. 2 ≠ 1

A[1] ⟷ A[2]

MAX-HEAPIFY (A, 2)



(Un



MAX-HEAPIFY (A, 2)

1. l = 4

2. r = 5

3. If 4 ≤ 8 and 10 > 9 (True)

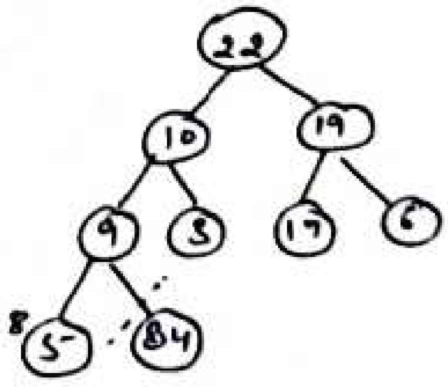
largest ← 4

If 5 ≤ 8 and 3 > 10 (False)

If 4 ≠ 2

A[2] ⟷ A[4]

MAX-HEAPIFY (A, 4)



MAX-HEAPIFY (A, 4)

1. l = 8

2. r = 9

3. If 8 ≤ 8 and 5 > 9 (False)

4. If 9 ≤ 8 (False)

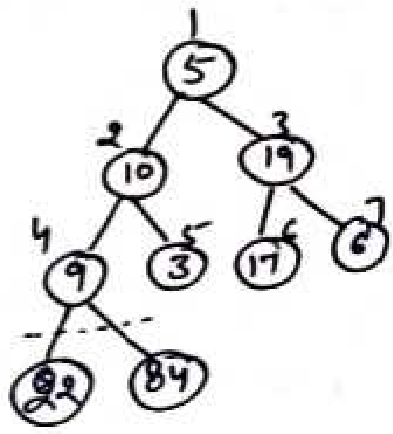
5. 4 ≠ 4 (False)

for $i = 8$ down to 2

A[1] ⟷ A[8]

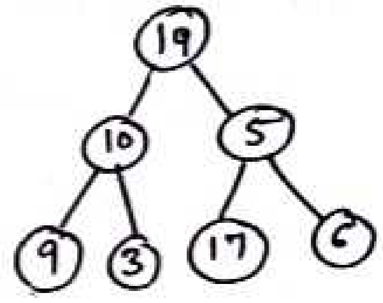
heap-size[A] ← 7

MAX-HEAPIFY(A,1)



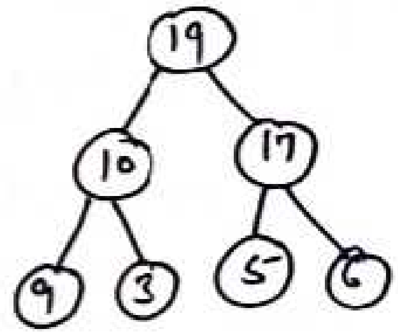
MAX-HEAPIFY (A,1)

1. $l = 2$
2. $r = 3$
3. If $2 \le 7$ and $10 > 5$ (True)
   largest ← 2
4. If $3 \le 7$ and $19 > 10$ (True)
   largest ← 3
5. If $3 \ne 1$
   A[1] ⟷ A[3]
   MAX-HEAPIY(A,3)



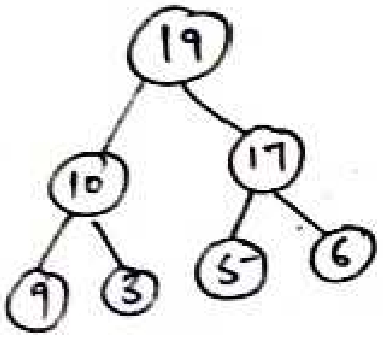
MAX-HEAPIFY (A,3)

1. $l = 6$
2. $r = 7$
3. If $6 \le 7$ and $17 > 5$ (True)
   largest ← 6
4. If $7 \nleq 7$ and $6 > 17$ (False)
5. If $6 \ne 3$
   A[3] ⟷ A[6]
   MAX-HEAPIFY(A, 6)



MAX-HEAPIFY (A, 6)

1. $l = 2i = 12$
2. $r = 2i+1 = 13$
3. If $12 \le 7$ (False)
4. If $13 \le 7$ (False)
5. $6 \ne 6$ (False)

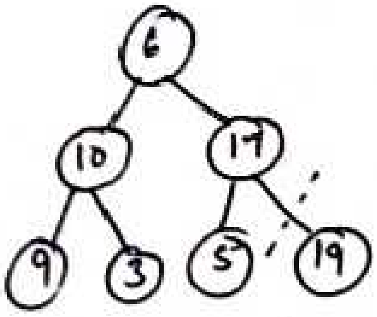


→ for $i = 7$ down to 2

exchange $A[1] \longleftrightarrow A[7]$

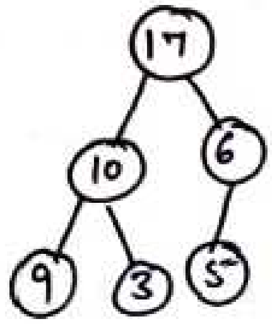
heap-size[A] $\leftarrow 7-1 = 6$

MAX-HEAPIFY (A, 1)



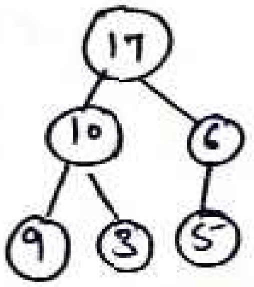
MAX-HEAPIFY (A, 1)

1. $l = 2$
2. $r = 3$
3. If $2 \le 6$ and $10 > 6$ (False)
   ~~else~~ largest $\leftarrow$ 2
4. If $3 \le 6$ and $17 > 10$ (False)
   largest $\leftarrow$ 3
5. If $3 \ne 1$
   $A[1] \longleftrightarrow A[3]$
   MAX-HEAPIFY (A, 3)



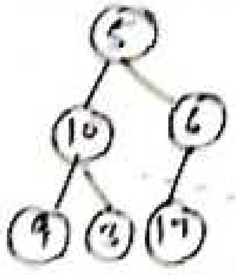
MAX-HEAPIFY (A, 3)

1. $l = 6$
2. $r = 7$
3. If $6 \le 6$ and $5 > 6$ (False)
   ~~then~~ ~~largest~~
   else largest $\leftarrow$ 3
4. If $7 \le 6$ (False)



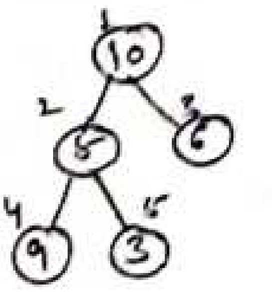
3 ≠ 3 (False)

```
for i = 6 down to 2
        exchange A[1] ⟷ A[6]
         heap-size [A] ← 6 - 1 = 5
            MAX-HEAPIFY (A,1)
```



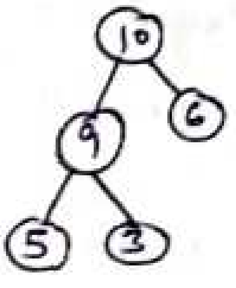
MAX-HEAPIFY (A,1)

```
1. l = 2
2. r = 3
3. If 2 ≤ 5 and 10 > 5 (True)
            largest ← 2.
4) If 3 ≤ 5 and 6 > 10 (False)
5. 2 ≠ 1
        A[1] ⟷ A[2]
        MAX-HEAPIFY (A, 2)
```



MAX-HEAPIFY (A, 2)

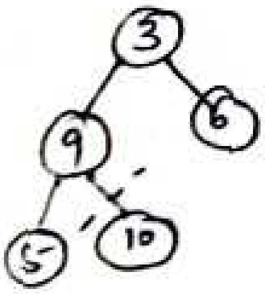
```
1. l = 4
2. r = 5
3. If 4 ≤ 5 and 9 > 5 (True)
        largest ← 4
4. If 5 ≤ 5 and 3 > 9 (False)
5. 4 ≠ 2
        A[2] ⟷ A[4]
        MAX-HEAPIFY (A, 4)
```



```
MAX-HEAPIFY (A,4)
1. l = 8    } (False)
2. r = 9
```

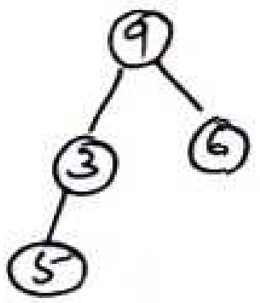
that

```
→ for i = 5 down to 2
        exchange A[1] ⟷ A[5]
            heap-size [A] ← 4
```



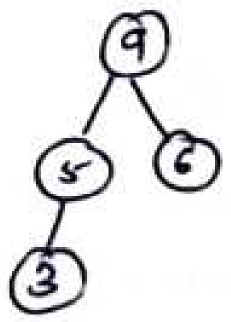
MAX-HEAPIFY(A,1)

```
1. l = 2
2. r = 3
4. If 2 ≤ 4 and 9 > 3 (True)
        largest ← 2
5. If 3 ≤ 4 and 6 > 9 (False)
6  2 ≠ 1
        A[1] ⟷ A[2]
        MAX-HEAPIFY(A, 2)
```



MAX-HEAPIFY(A,2)

```
1. l = 4
2. r = 5
3. If 4 ≤ 4 and 5 > 3 (True)
        largest ← 4
4. If 5 ≤ 4 (False)
5.  4 ≠ 2   A[2] ⟷ A[4]
            MAX-HEAPIFY (A, 4)
```

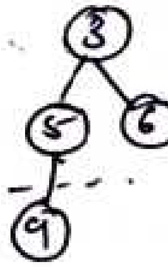


MAX-HEAPIFY(A,4)

```
1. l = 8
2. r = 9              } false.
3. If 8 ≤ 4 (False)
```

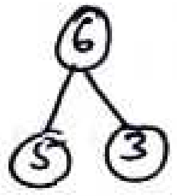
the bir
The
low

```
for i = 4 down to 2
        exchange A[1] ⟷ A[4]
        heap-size[A] ← 4-1 = 3
        MAX-HEAPIFY(A, 1)
```



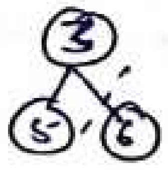
MAX-HEAPIFY(A,1)

```
1. l = 2
2. r = 3
3. If 2 ≤ 3 and 5 > 3 (True)
        largest ← 2
4. If 3 ≤ 3 and 6 > 5 (True)
        largest ← 3
5. 3 ≠ 1
        A[1] ⟷ A[3]
        MAX-HEAPIFY(A,3)
```



MAX-HEAPIFY(A,3)

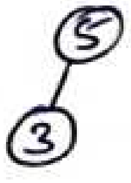
```
1. l = 6
2. r = 7
3. If 6 ≤ 3 (False)
      else largest ← 3
4. If 7 ≤ 3 (False)
5. If 3 ≠ 3   (False)
→ for i = 3 down to 2
        exchange A[1] ⟷ A[3]
        Heap-size ← 2.
        MAX-HEAPIFY(A,1)
```



MAX-HEAPIFY(A,1)

```
1. l = 2
2. r = 3
```

3. If $2 \leq 2$ and $5 > 3$ (True)
   largest $\leftarrow 2$

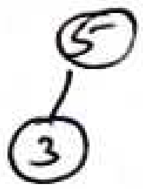


4. If $3 \leq 2$ (False)

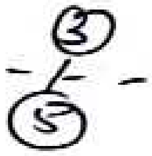
5. If $2 \neq 1$
   $A[1] \leftrightarrow A[2]$
   MAX-HEAPIFY (A,2)

MAX-HEAPIFY (A,2)

1. $l = 4$
2. $r = 5$
3. If $4 \leq 2$ (False)
   else largest $\leftarrow 2$
4. If $5 \leq 2$ (False)
5. If $2 \neq 2$ (False)



→ for $i = 2$ down to 2
   exchange $A[1] \leftrightarrow A[2]$
   heapsize $\leftarrow 1$
   MAX-HEAPIFY (A,1)



MAX-HEAPIFY (A,1)

1. $l = 2$
2. $r = 3$
3. If $2 \leq 1$ (False)
   else largest $\leftarrow 1$
   If $3 \leq 1$ (False)
   If $1 \neq 1$ (False.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 5 | 6 | 9 | 10 | 17 | 19 | 22 | 84 |

Sorted

## Analysis of Heapsort

In HeapSort algorithm, two functions are called [illegible]

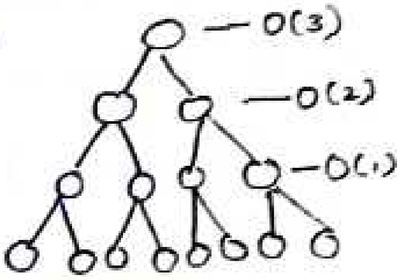
(a) BUILD_MAX_HEAP()
(b) MAX_HEAPIFY()

### (a) Analysis of BUILD_MAX_HEAP()

This function is used to create a MAX_HEAP().

Maximum number of nodes in a heap at height h can be $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

If MAX-HEAPIFY function is applied to any of the node of height h would be $O(h)$



So, Total amount of work done, If the MAXHEAPIFY function is applied to all the nodes $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ of height h would be

$$= \sum_{h=0}^{\lg n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$= \sum_{h=0}^{\lg n} \left\lceil \frac{n}{2^h \cdot 2} \right\rceil (ch)$$

$$= \frac{cn}{2} \left[ \sum_{h=0}^{\lg n} \left( \frac{h}{2^h} \right) \right]$$

If I write

$$= \frac{cn}{2} \left[ \sum_{h=0}^{\infty} \frac{h}{2^h} \right] \qquad \left[ \sum_{h=0}^{\infty} \frac{h}{2^h} > \sum_{h=0}^{\lg n} \frac{h}{2^h} \right]$$

So we consider this above equation in which h varies from 0 to ∞

The solution of $\left[\sum_{h=0}^{\infty} \frac{h}{2^h}\right]$ is 2.

↳ Harmonic Progression Series after Solving this series by applying Differentiation and Integration we will get 2.

$$= \frac{cn}{2} \times 2$$

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

Analysis of MAX-HEAPIFY

In order to do one swapping, and to move just 1 level down I have to do 2 comparisions.

So, Total number of comparisions required

$$= (2 * \underbrace{\log n}_{\text{height of the tree}})$$

$$= O(\log_2 n)$$

Analysis of HeapSort

$$= O(n) + (n-1)\, O(\lg n)$$
$$= O(n) + O(n\lg n) - O(\lg n)$$
$$= O(n\lg n)$$ Proved.

$\boxed{5,9,10,6,3,2,1}$

$j=n-1$ | $j=n-2$ | $j=n-3$ | [illegible] | $j=1$ | $j=0$

$= 0+1+2+3+\dots+n-1$

$= (n-1)(n-$ [illegible] $, O(n^2)$

# SELECTION SORT

1. The idea behind the selection sort is that we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array.

2. Assume that we wish to sort the array in increasing order. we begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element.

3. We then reduce the effective size of the array by one element and repeat the process on the smaller sub array. The process stop when the effective size of the array becomes 1.

```
Selection - Sort (A)
1.  n ← length[A]
2.  for j ← 1 to n-1
3.      Smallest ← j
4.      for i ← j+1 to n
5.          if A[i] < A[Smallest]
6.              Smallest ← i
7.      exchange (A[j], A[smallest])
```

Example  Sort the elements using Selection Sort
A[ ] = { 5, 2, 1, 4, 3 }

A =

| | | | | |
|---|---|---|---|---|
| 5 | 2 | 1 | 4 | 3 |

length[A] = 5
n ← length[A] ← 5

→ for j = 1 to 4

```
    smallest ← 1
    for i = 2 to 5
        if A[2] < A[1]
            2 < 5    (True)
            Smallest ← 2

    for i = 3 to 5
        if A[3] < A[2]
            1 < 2    (True)
            Smallest ← 3

    for i = 4 to 5
        if A[4] < A[3]
            4 < 1    (False)
```

```
- for i = 5 to 5
      if A[5] < A[3]
           3 < 1   (False)
  exchange ( A[1], A[3])
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 | 2 | 5 | 4 | 3 |

```
→   for j = 2 to 4
        Smallest ← 2
        for i ← 3 to 5
            if A[3] < A[2]
                 5 < 2    (False)
        for i ← 4 to 5
            if A[4] < A[2]
                 4 < 2 ( False)
        for i ← 5 to 5
            if A[5] < A[2]
                 3 < 2   (False)
        exchange (A[2], A[2])
```

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 5 | 4 | 3 |

```
→   for j = 3 to 4
        Smallest ← 3
        for i = 4 to 5
            if A[4] < A[3]
                 4 < 5 (True)
                 Smallest ← 4
        for i = 5 to 5
            if A[5] < A[4]
                 3 < 4 (True)
                 Smallest ← 5
        exchange (A[3], A[5])   A[3] ↔ A[5]
```

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |

```
→ for j ← 4 to 4
      smallest ← 4
      for i ← 5 to 5
          if A[5] < A[4]
              5 < 4 ( False
      exchange (A[4], A[4])
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|

## ANALYSIS OF SELECTION SORT

Selection Sort is very easy to analyze since none of the loops depends on the data in the array. Selecting the lowest elements requires scanning all elements (this takes (n-1) compatisions) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining (n-1) elements and so on, for a total of

$$(n-1)+(n-2)+ \text{-----} +2+1 = \theta(n^2)$$

Each of these scans requires one swap for a total of n-1 swaps. (final element already in place)

$$\theta(n^2)+\theta(n) = \theta(n^2)$$

Thus, Comparsions dominate the running time, which is $\theta(n^2)$

* The Best case complexity of Selection Sort is $O(n^2)$ and also worst case complexity of Selection Sort is $O(n^2)$

# INSERTION SORT

1. Insertion sort technique is similar to the technique of playing with cards. It is the in-place sorting algorithm that means it requires constant amount of extra memory space.

2. Every iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input.

## Algorithm

```
Insertion_Sort(A)
1.  for j ← 2 to length[A]
2.    { do  key ← A[j]
3.         //insert A[j] into sorted sequence A[1 .. j-1]
4.         i ← j-1
5.         while ( i>0 and A[i] > key)
6.              A[i+1] ← A[i]
7.              i ← i-1
8.         A[i+1] ← key
    }
```

example :-

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 8 | 5 | 2 | 9 |

→ length[A] = 4

→ for j=2 to 4

key ← A[j]

$\boxed{key \leftarrow 5}$

i = 1

while ( 1>0 and A[1] >5 )

A[2] ← A[1] $\boxed{A[2] \leftarrow 8}$

i = 0

while (0>0) false

$\boxed{A[1] \leftarrow 5}$

|   |   |   |   |
|---|---|---|---|
| 5 | 8 | 2 | 9 |

```
→ for j=3 to 4
    Key ← A[3]
    key ← 2
    i=2
    while(2>0 & 8>2)
        A[3] ← A[2]
        A[3] ← 8        i=1
    while(1>0 & 5>2)
        A[2] ← 5
        i=i-1 =0
```

$$\begin{array}{|c|c|c|c|} 1 & 2 & 3 & 4 \\ \hline 5 & 8 & 2 & 9 \\ \hline \end{array}$$

A[1] ← 2

$$\begin{array}{|c|c|c|c|} 1 & 2 & 3 & 4 \\ \hline 2 & 5 & 8 & 9 \\ \hline \end{array}$$

```
→ for j=4 to 4
    Key ← A[4]     key←9
    i=3
    while( 3>0 & 8>9) False
    [A[4] ← 9]
```

$$\begin{array}{|c|c|c|c|} \hline 2 & 5 & 8 & 9. \\ \hline \end{array}$$

## Analysis of Insertion Sort

To calculate the running time, we check

= How many comparisions are required + How many movements are required.

<u>In Worst case</u>:- ( Input is in reverse order)

|  |  | comparision | movement |  |  |
|---|---|---|---|---|---|
| If j=2 | —— | 1 + | 1 | =2 | ⇒ 2×1 |
| j=3 | —— | 2 + | 2 | =4 | ⇒ 2×2 |
| j=4 | —— | 3 + | 3 | =6 | ⇒ 2*3 |
| j=5 | —— | 4 + | 4 | =8 | ⇒ 2*4 |

$j = n$ — $(n-1) + (n-1) = 2(n-1)$

$= 2(1) + 2(2) + \ldots\ldots + 2(n-1)$

$= 2[1+2+3+\ldots\ldots n-1]$

$= \dfrac{2(n-1) \times n}{2} = O(n^2)$

<u>In Best case:-</u> The sorted list is passed as input

(1 2 3 4 5 6 7 8 9)

| | comparision | | movement | |
|---|---|---|---|---|
| j = 2 | 1 | + | 0 | $\rightarrow$ 1 |
| j = 3 | 1 | + | 0 | $\rightarrow$ 1 |
| j = 4 | 1 | + | 0 | $\rightarrow$ 1 |

⋮

∴ (n-1) comparisions + 0 movements Hence the best case complexity is $\Omega(n)$

<u>In Avg case:-</u> It is nearly an order of $n^2$ i.e $O(n^2)$

**SHELL SORT**

<u>Algorithm.</u>

<u>Input:-</u> An array a of length n with array elements numbered 0 to n-1.

```
1.  inc ← round (n/2)
2.  while inc > 0
3.      { for i = inc to n-1.
```

```
    {
        temp ← a[i]
        j ← i
        while j ≥ inc and a[j - inc] > temp
          {
              a[j] ← a[j - inc]
              j ← j - inc
          }
        a[j] ← temp
    }
4.  inc ← round (inc/2.2)
  }
```

<u>example</u>:- Sort the elements of the given array A using shell sort algorithm.

A = {20, 35, 18, 41}

<u>Solution</u>:- 4 numbers to be sorted, shell's increment will be floor($n/2$) i.e floor($4/2$) ⇒ floor(2) = 2

```
inc → 2
while 2>0
for i = 2 to 3
    {  temp ← a[2]
       temp ← 18
       j ← 2
        while (j ≥ 2 and a[2 - 2] > 18)
          while (2 ≥ 2 and 20 > 18)
            {
                  a[2] ← a[2 - 2]
                  a[2] ← a[0] ⇐ · a[2] ← 20
                  j ← 2 - 2
                  j ← 0
            }
        while (0≥2) False
        a[0] ← 18
    }
for i = 3 to 3
    {  temp ← 41
        j ← 3
        while (3≥2  and a[3-2] > 41)
        While (3≥2  and a[1] >41)
        while ( 3≥2  and  35>41) false)
        {
        }
        a[3] ← 41
    }
 inc ← round (2/2)
 inc ← 1
while 1>0
    for i=1  to 3
        {  temp ← a[1]
           temp ← ·35
           j ← 1
           While (1≥1  and  a[1 - 1] > 35)
           While  (1≥1  and 18>35)  False
              a[1]← 35

        }
    for i = 2 to3
        { temp ← a[2] = 20
             j= 2
              while 2≥1  and  a[1] > 20
              while 2≥1  and  35>20  (True)
```

|  |  |  |  |
|---|---|---|---|
| 18 | 35 | 20 | 41 |

```
        a[2] ← a[2 -1]
        a[2] ← a[1] = 35
    a[1] ← temp ← 20                          | 10 | 20 | 35 | 41 |
 for i=3 to 3
    {  temp ← a[3]
       temp ← 41
        j ← 3
         while 3≥1 and a[2] > 41
         while 3≥1 and 35 ≥ 41 False
          ~~while 3≥1 and af~~.
         a[3] ← 41
    }
  inc ← round(1/2)
      ← 0
  while 0>0 (False)
```

Sorted list | 10 | 20 | 35 | 41 |

Analysis of Shell Sort

1. Shell sort improves on the efficiency of insertion sort by quickly shifting values to their destination.
2. In this method, instead of sorting the entire array at once, the array is first divided into smaller segments. Then these segments are sorted seperately using the insertion sort.
3 The shell sort is also called diminishing increment sort because the value of k continuously decreases, where k is preferably a prime number

Best case Complexity of Shell Sort-

The best case in the shell sort is when the array is already sorted in the right order. The number of comparisions is less, So the best case complexity of shell sort is O(n)

Worst case complexity of Shell Sort-

The worst case time complexity of shell sort depends on the increment sequence. For increments 1 4 13 40 121 ----. which is what is used here, the time complexity is $O(n^{3/2})$. For other increments, time complexity is known to be $O(n^{4/3})$ and even $O(n \cdot lg^2(n))$.

| omparing Techniques | Bubble Sort | Selection Sort | Insertion Sort | Shell Sort | Quick Sort | Merge Sort | Heap Sort |
|---|---|---|---|---|---|---|---|
| . Best case complexity | $n$ | $n^2$ | $n$ | $n$ | $n\log n$ | $n\log n$ | $n\log n$ |
| . worst case complexity. | $n^2$ | $n^2$ | $n^2$ | $n\log^2 n$ | $n^2$ | $n\log n$ | $n\log n$. |
| . Average case complexity | $n^2$ | $n^2$ | $n^2$ | $n\log^2 n$ | $n\log n$ | $n\log n$ | $n\log n$. |
| 4. Stability of Algorithms | stable | Not Stable | Stable | Not stable | Stable | stable | Not Stable |
| 5. Method adopted by Algorithm. | Swapping method | Selection Method. | Insertion Method. | Insertion Method | Partioning Method (Divide + conquer Technique) | Divide + conquer. Technique. | Selection method. |