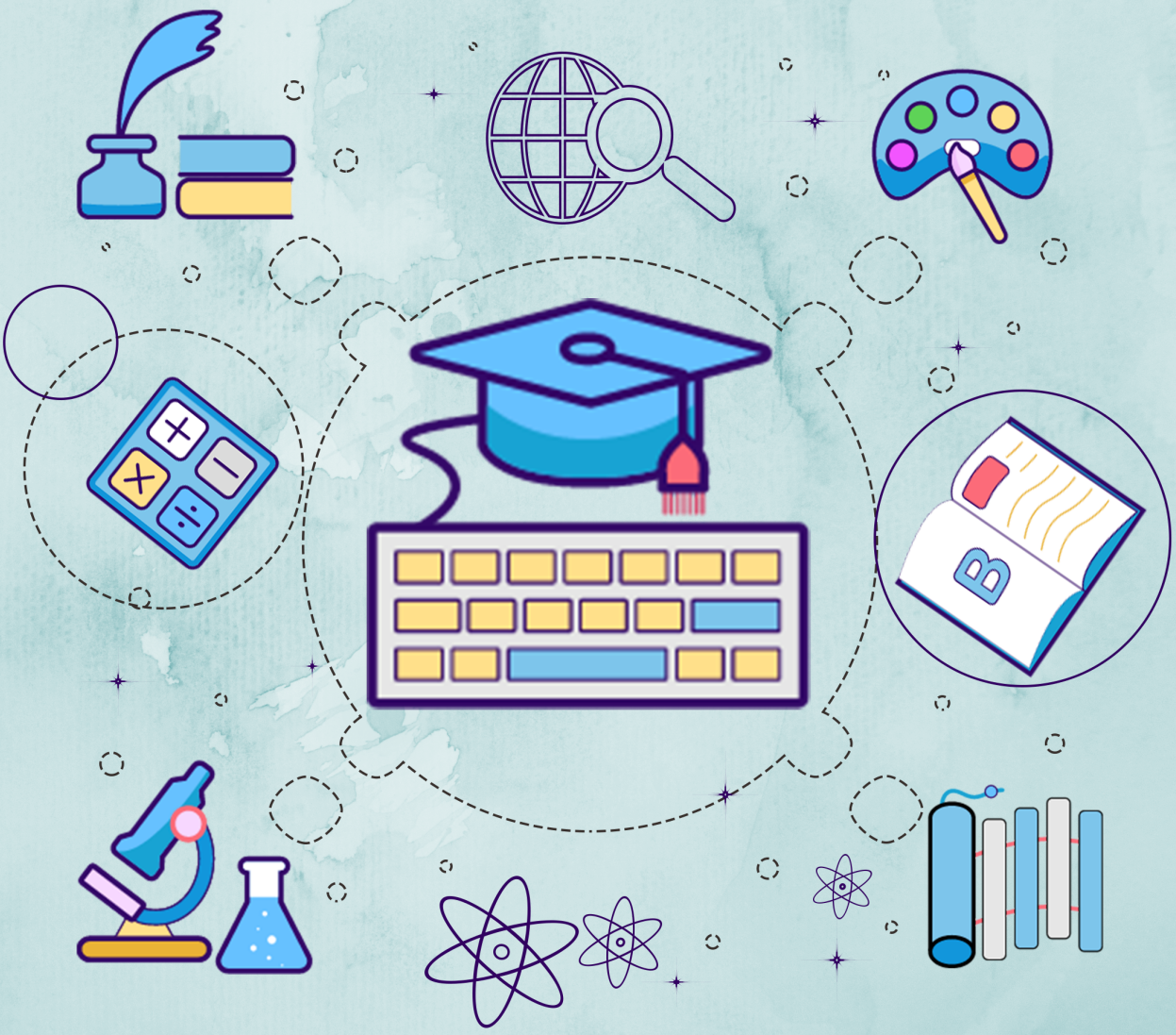
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

ALGORITHM ANALYSIS AND DESIGN

CST 306

Module 1

Related Link :

- KTU S6 CSE NOTES 2019 SCHEME
- KTU S6 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S6 CSE SOLVED
- KTU CSE TEXTBOOKS S6 B.TECH PDF DOWNLOAD
- KTU S6 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

Module 1.

* Algorithm is a finite set of instructions to solve a given computation problem.
* All algorithms must satisfy the fol. criteria (Properties of algorithm)

1. Input :- An algorithm accepts zero or more i/ps.
2. Output :- An algorithm should produce one or more o/ps.
3. Finiteness :- An algorithm should be finite, it means it should terminate in a finite no. of steps.
4. Uniqueness :- Each step of an algorithm is uniquely defined. The o/p of a particular step depends on the i/p and o/p of processing steps.
5. Unambiguous :- Each instruction of an algorithm should be clear and have only one interpretation.
6. Generality :- The algorithm should be general, ie it can be applied to a sequence of i/ps.

Recursive Algorithms.

A recursive fn is a fn that is defined in terms of itself. An algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is direct recursive. Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

Any algorithm that can be written using assignment, the if-then-else stmt and while stmt can also be written using assignment, the if-then-else stmt & recursion.

eg :- To find the factorial of a number.

Analysis of algorithms

It is the process of estimating the resources required to run an algorithm. The major resources are 1. Space 2. Time.

Space Complexity

It is the estimation of memory required to run an algorithm.

The space needed by an algorithm is seen to be the sum of the fol. components.

1. A fixed part that is independant of the characteristics (eg :- number, size) of the inputs and outputs. This part typically includes the

instruction space (ie space for the code), space for simple variables, & fixed size component variables, space for constants & so on.

2. A variable part that consists of the space needed by component variables whose size is dependant on the particular problem instance being solved, the space needed by the reference variable and the recursion stack space.

∴ The space requirement $S(P)$ of any alg P may be written as $S(P) = C + Sp$, where C is the constant part & Sp is the variable pa

Time Complexity

It is process of estimating the amount of computer time required to run an algorithm. It is the sum of compile time & run time.

There are 3 types of time complexity.

(a) Best case:- minimum amount of time reqd to run an algorithm.

(b) Average case:- Avg. amount of time reqd to run an algorithm.

(c) Worst case:- Maximum amount of time reqd to run an algorithm.

The time complexity of an alg. is directly proportional to the size of the i/p.

We use a set of notations for representing the time complexity of an algorithm. Such notations are called asymptotic notations.

Asymptotic Notations:-

The word Asymptotic means approaching a value or curve arbitrarily closely (ie as some sort of limit is taken).

When we need to analyse the complexity of any algorithm in terms of time & space, we can never provide an exact no. to define the time & space required by the algorithm, instead we express it using some standard notations, also known as Asymptotic notations.

Asymptotic Notations allow us to analyze an algorithm's running time by identifying its behaviour as the i/p size for the algorithm increases. This is also known as an algorithm's growth rate

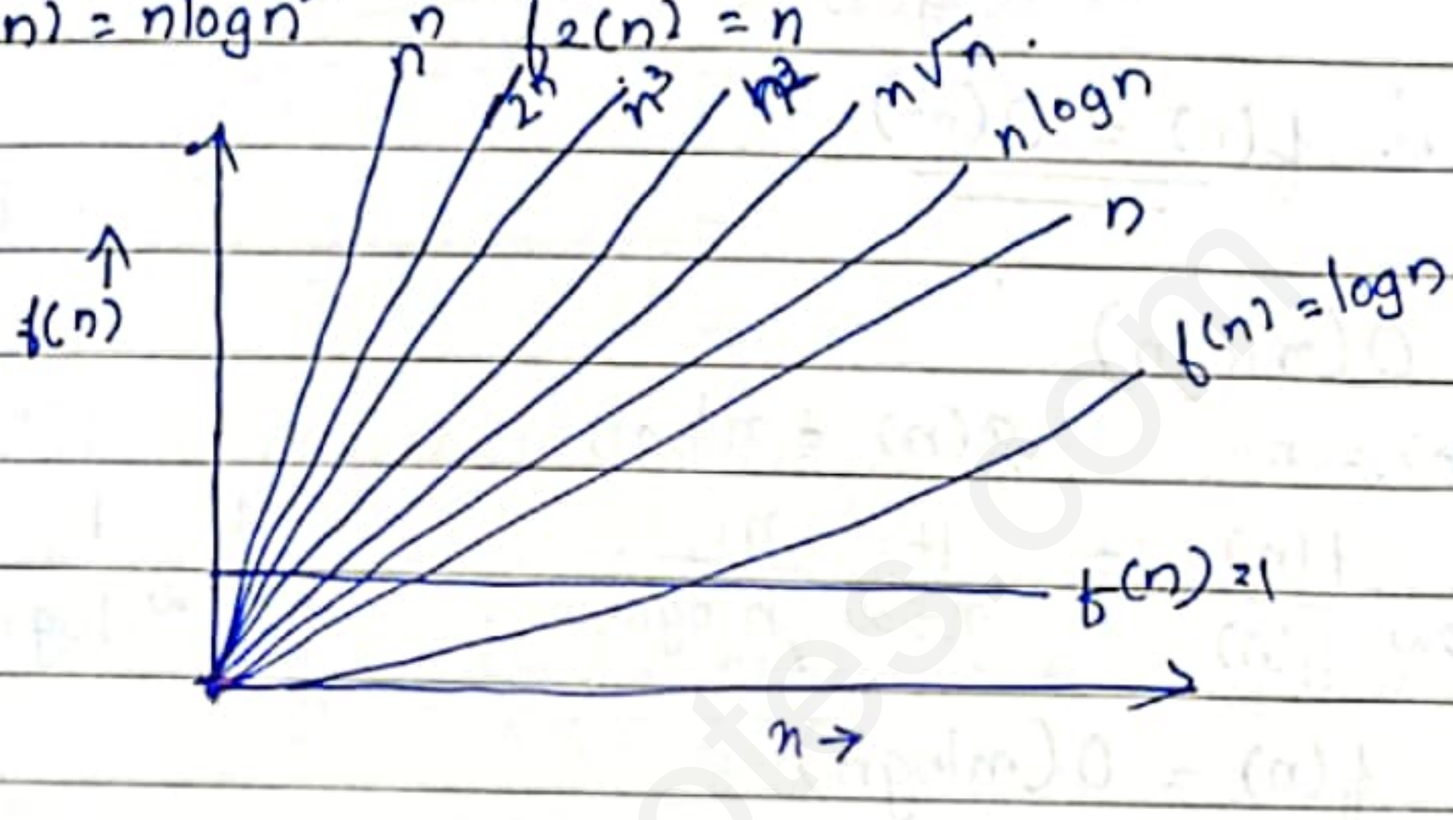
eg:- $f_1(n) = 20n^2 + 3n - 4$ $f_2(n) = n^3 + 100n - 2$ Which fn grows faster?
For $f_1(n)$, the growth of the fn will depend on n^2 & for $f_2(n)$ n^3 the growth of fn depends on n^3. So $f_2(n)$ will grow faster than $f_1(n)$. Here we ignore the constants & ie 20 in $20n^2$. and insignificant parts of the expressions ($3n-4$ and $100n-2$).

Eg:- 1. $f_1(n) = n^2$ $f_2(n) = 10n$

2. $f_1(n) = n\log n$ $f_2(n) = n$

3. $f_1(n) = n^n$ $f_2(n) = 2^n$

4. $f_1(n) = n\log n$ $f_2(n) = n$

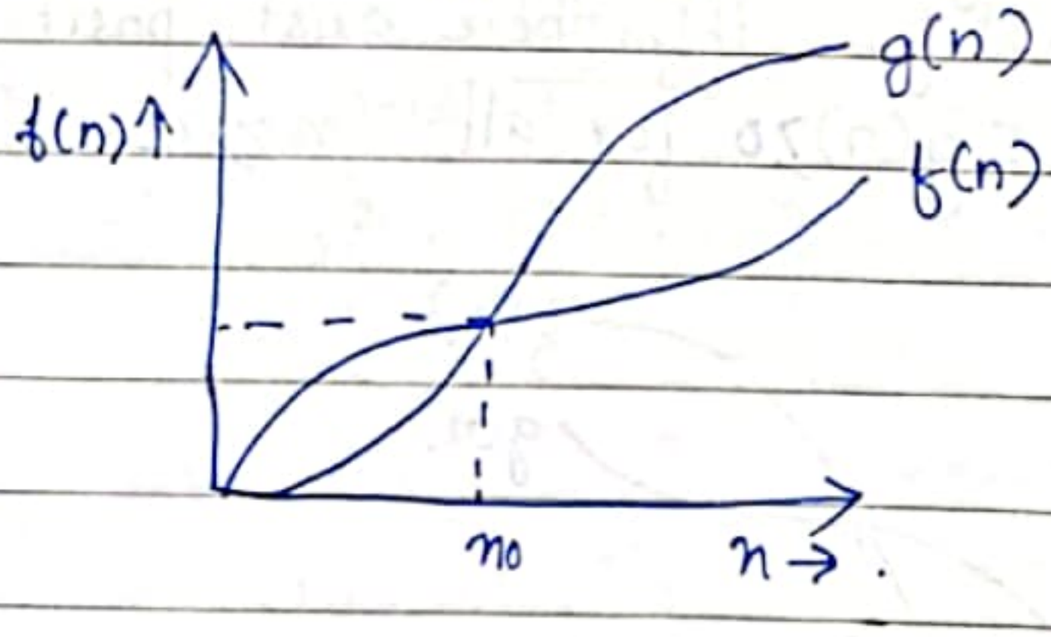


There are five asymptotic notation

(1) Big-Oh (O) (2) Big-Omega (Ω) (3) Big-theta (Θ)
(4) Little-Oh (o) (5) Little-Omega (ω).

Big-Oh Notations (O)

Let f and g are two non-negetive fn, ~~f=c~~ $f(n) = O(g(n))$
If there exists a positive integers n_0 and a positive constant c
such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.



The Big-Oh notation is used to represent the worst case complexi
of an algorithm. It provides us with an asymptotic upper

bound for the growth rate of runtime of an algorithm.

eg:- $f(n) = 3\log n + 100$

The ~~big-Oh notation~~ is used to represent

Definition by using L- Hospital rule.

$f(n) = Og(n)$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

(i) eg:- Check $n^2+n = O(n^2)$

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

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2+n}{n2} = \frac{\infty}{\infty} = 1$

$\therefore f(n) = O(n^2)$

(ii) check $n = O(n\log n)$

$f(n) = n \qquad g(n) = n\log n$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{n\log n} = \lim_{n\to\infty} \frac{1}{\log n} = \lim_{n\to\infty} \frac{1}{\infty} = 0$

$\therefore f(n) = O(n\log n)$

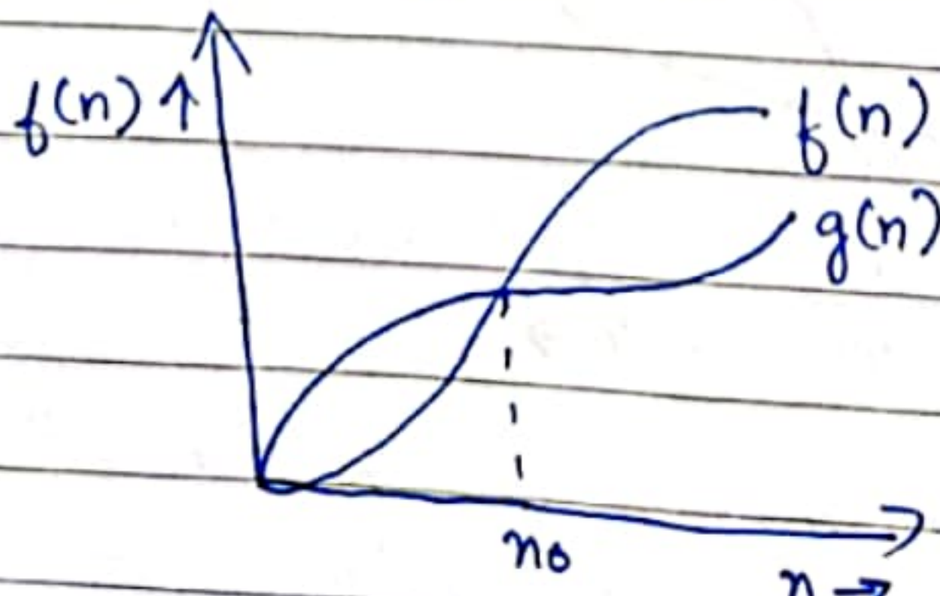
$f(n) \le cg(n)$

$n = O(n\log n) \qquad n = O(n^2) \qquad n = O(n^3)$

$n = O(n^4) \qquad n = O(n)$

Big Omega (Ω) notation (Best case)

Let f and g be the 2 non-negetive $fn \ge \Omega g(n)$ if the set of functions f, iff there exist positive constants c, no such that $f(n) \ge c\,g(n) \ge 0$ for all $n \ge n_0$.

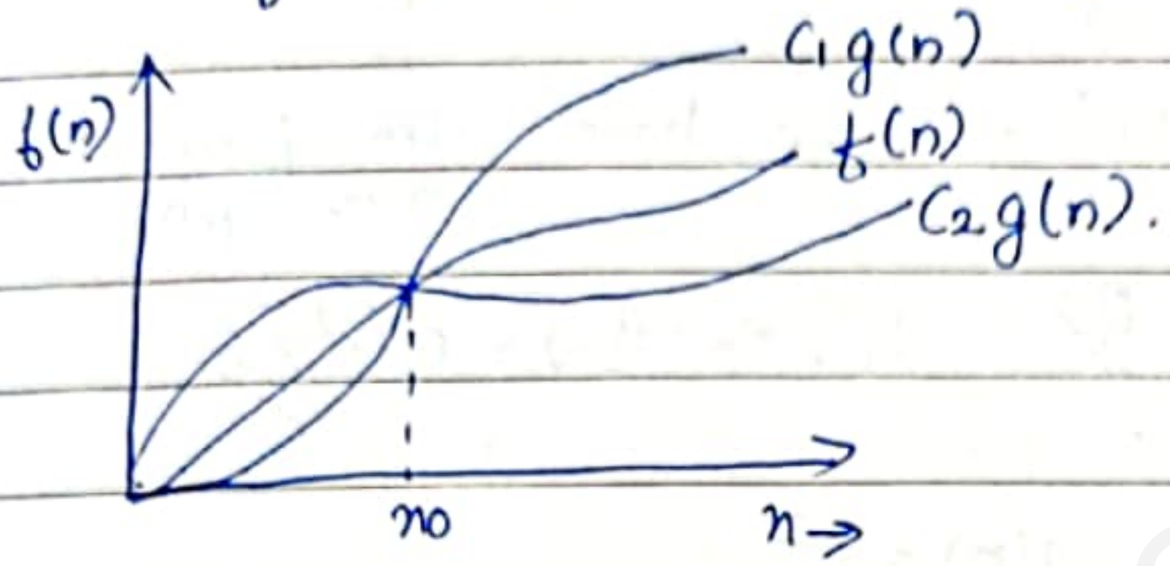


$n = \Omega(n)$ $\quad n = \Omega(\log n)$ $\quad n = \Omega(1)$ $\quad n \neq \Omega(n^2)$

Definition by using L-hospital's rule $f(n) = \Omega g(n)$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$

Big theta notation (Θ) Big theta notation (Θ)

Let f & g are two non-negelive fns, then $f(n)$ is $g(\Theta(g(n))$ if for some real constants c_1, c_2 & n_0 ($c_1 > 0$, $c_2 > 0$, $n_0 > 0$), ~~$c_1 g(n)$ is $f \leq f(n)$~~ $c_2 g(n) \leq f(n) \leq c_1 g(n)$ for all $n > n_0$



Θ is used to represent the average case complexity

Definition by using L-Hospital rule.

$f(n) = \Theta g(n)$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is a +ve constant.

eg:- $n^2 + n = \Theta(n^2)$ $\quad n^2 = \Theta(100n^2)$,

eg: Compare n^2 and n^2+6n.

$f(n) = n^2$ $\quad g(n) = n^2+6n$.

$\lim_{n\to\infty} \frac{n^2}{n^2+6n} = \lim_{n\to\infty} \frac{\infty}{\infty} = 1$ $\quad \therefore f(n) = \underline{\Theta(g(n))}$.

(2) Compare $\log n$ and $\log n^2$

$f(n) = \log n$ $\quad g(n) = \log n^2$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log n}{\log n^2} = \lim_{n\to\infty} \frac{\log n}{2\log n} = \frac{1}{2}$

$\therefore \log n^2 = \Theta(\log n)$.

Other examples.

A. All cubic fns, such as n^3, $5n^3+4n$, $105n^3+4n^2+6n$, have same rate of growth: $\Theta(n^3)$

B. All quadratic fns, such as n^2, $5n^2+4n+6$, n^2+5 have same rate of growth: $\Theta(n^2)$

C. All logarithmic fns, such as $\log n$, $\log n^2$, $\log(n+n^3)$, have same rate of growth: $\Theta(\log n)$

Little - Oh Notation (o)

Let f and g are two non-negetive functions, $og(n)$ is the set of functions iff there exists two +ve constants c, no such that $0 \leq f(n) < c\,g(n)$.

Little-O means loose upper bound of $f(n)$.

Using L-Hospital's rule.

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

Q check $7n+8 = o(n^2)$

Ans Using L-Hospital's rule we have $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{7n+8}{n^2}$

$= \lim_{n\to\infty} \frac{7}{2n} = 0$ // $\therefore f(n) = o(n^2)$ //

Q. Compare n & n^2.

Let $f(n) = n$ $g(n) = n^2$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{n^2} = \lim_{n\to\infty} \frac{1}{n} = 0$ //

$\therefore n = o(n^2)$ //

Q. Compare n & logn.

Let $f(n) = n$ $g(n) = \log n$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{\log n} = \infty$ hence $n = \omega(\log n)$ //

Little - Omega notation (ω)

Let f and g are two non-negetive functions. $\omega g(n)$ is the set of functions iff there exist a two +ve constants c, no such that $f(n) > cg(n) \geq 0$

Using L-Hospital's rule.

$f(n) = \omega g(n)$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Big-Oh (O) $\Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	Little Oh (o) $\Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
Big-Omega (Ω) $= \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$	Little Omega (ω) $\Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
Big Theta (Θ) $= \lim_{n\to\infty} \frac{f(n)}{g(n)} = c$.	

Big-Oh $\Rightarrow 0 \leq f(n) \leq cg(n)$ | little-Oh $\Rightarrow 0 \leq f(n) < cg(n)$

$\Omega \Rightarrow f(n) \geq cg(n) \geq 0$ | $\omega \Rightarrow f(n) > cg(n) \geq 0$

$\Theta \Rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

Eg: Analysis of complexity of Linear search algorithm.

Linear search.

Given an array of n elements and a key, the linear search algorithm check whether the key is present or not in the array. If the key is not present the algorithm returns -1. Otherwise the alg. returns the position of the key.

```
Linear Search (A[], key)
    for(i=0; i<n; i++)
        if(A[i] == key)
            return i+1;
    return -1;
```

Complexity of Linear Search.

Best case complexity :- The best case occurs when the key is present at first position. No. of comparisons reqd is one.

So best case complexity is $\Omega(1)$

Worst Case Complexity:- Worst case complexity occurs when the key is present at the last position or the key is not present.

No. of comparisons reqd is n.

∴ The worst case complexity is $O(n)$

Average Case Complexity

If the key is present at first posn, comparison reqd is one.

If the key is present at second posn, comparison reqd is two.

If the key is present at third posn comparison reqd is three & so on

So the avg. no of comparisons $= \frac{1+2+3+\cdots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$

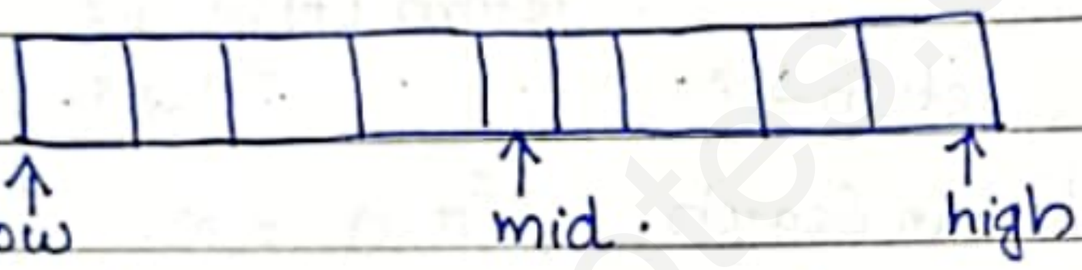
∴ Average Case Complexity is $\Theta(n)$.

Analysis of Complexity of Binary Search Algorithm.

Given a sorted array and a key, the binary search finds the position of the key. If the key is present in the array other wise it will return -1. Binary search is based on divide and conquer approach. Divide and conquer approach consists of 3 steps.

1. Divide :- Divide the problem into subproblems.
2. Conquer :- Conquer the subproblem by sub problem by solving them recursively.
3. Combine : Combine the soln of subproblems to form the soln of original pblm.

Let A be a sorted array of n elements and key be the element to be searched, let low & high represents the first and last index position. Initially low = 0, high = n-1




```
Binary search (A[], key, low, high)
 { if (low > high)
        return -1
   else
        mid =(low + high) /2 .
        if (key = =A [mid])
                return mid+1
        if( key < A[mid])
                BinarySearch (A[], key, low, mid-1)
        else
                BinarySearch (A[], key, mid+1, high)
 }
```

The algorithm finds the mid posn in the array. and compares the middle element with the key. If both are same, the algorithm returns mid+1. If the key is less than middle element will search left subarray. Other wise will search right sub array. If the key is not present in the array, alg. returns -1.

Eg:-

2	7	9	11	14	16	18

* key = 11

low = 0, high = 6 mid = $\frac{low+high}{2}$ = $\frac{0+6}{2}$ = 3

compare a[mid] and key 11 & 11 equal.

return mid+1 => 4

* key = 16 low = 0 high = 6

mid = $\frac{low+high}{2}$ = $\frac{0+6}{2}$ = 3

if (key == A[mid]) 16 == 11 False

low = 4 high = 6

mid = $\frac{4+6}{2}$ = 5

if (mid == A[key]) = 16 == 16 Equal

return mid+1 => 6.

* key = 2.

low = 0 high = 6.

mid = $\frac{low+high}{2}$ = $\frac{0+6}{2}$ = 3

if (A[mid] == key) 11 == 2 - False.

if (key < A[mid]) 2 < 11

low = 0 high = 2.

mid = $\frac{0+2}{2}$ = 1

if (key == A[mid]) 2 == 7 - False

if (key < A[mid]) 2 < 7 - True.

low = 0 high = 0.

mid = $\frac{0+0}{2}$ = 0.

if (key == A[mid]) 2 == 2 → True.

return mid+1 => 1 //

* key = 17

low = 0 high = 6 mid = $\frac{0+6}{2}$ = 3

if (key == A[mid]) => 17 = 11 => False.

if (key < A[mid]) => 17 < 11 => False

if (key > A[mid]) => 17 > 11 => True low = 4 high = 6

mid = $\frac{4+6}{2}$ = 5. If (key == A[mid]) = 17 == 16 → False

low = 6, high = 6 mid = $\frac{6+6}{2}$ = 6. if (key == A[mid]) = 17 == 18 =>

low = 6 high = 5 ∴ Return -1

Complexity

Best case :- It occurs when the key is present at middle posn.
No. of comparisons required is one.
∴ The base case complexity is $\Omega(1)$

Worst Case Complexity

It occurs when the key is not present in the array. Let A contains n elements. After the first iteration $n/2$ elements. Second iteration $n/4$ elements. . . .

no. of comparison is $\frac{n}{2^d} \geq 1$

ie $n \geq 2^d$ $\quad \therefore d = \log n$.

The worst case complexity is $O(\log n)$ Avg case $= O(\log n)$
Space complexity is $O(1)$. ie the space taken by the alg is same for any no. of elements in the array.

Compare the growth rate of fol. fns.

(a) n^2 & 2^n

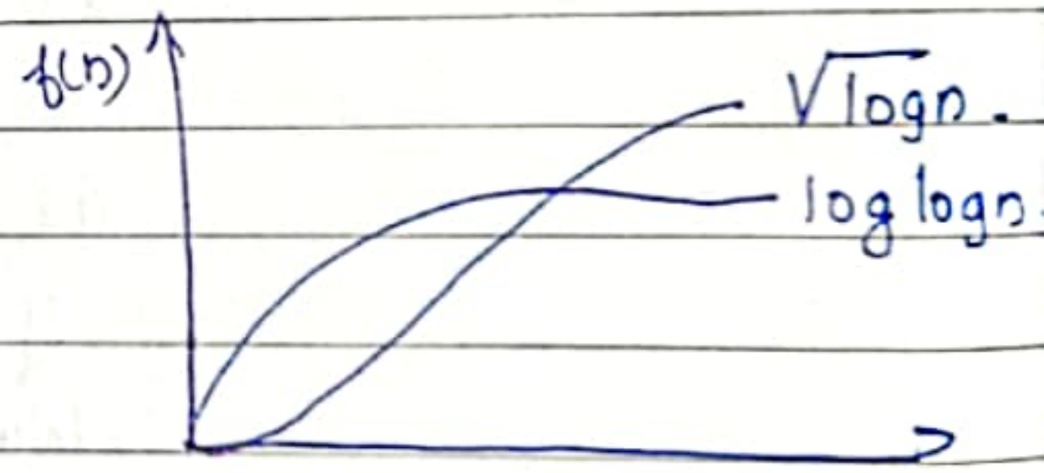
$2^n > n^2$ ∴ 2^n grows faster than n^2

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

(b) $\sqrt{\log n}$ and $\log \log n$.

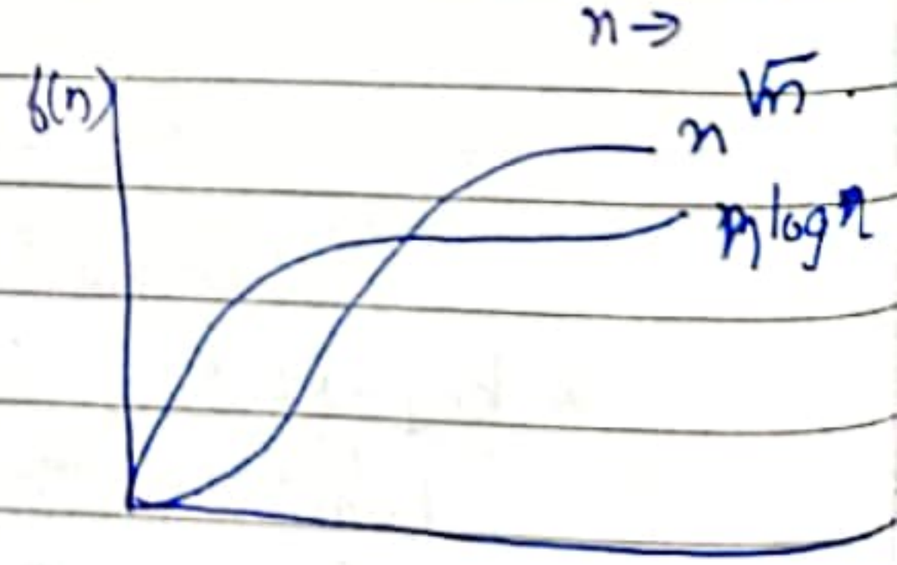
$\sqrt{\log n} > \log \log n$

∴ $\log \log n = O(\sqrt{\log n})$



(c) $n^{\sqrt{n}}$ & ~~$n \log n$~~ $\cdot n^{\log n}$

~~n~~ $n^{\log n} = O(n^{\sqrt{n}})$ //



We can classify algorithms into two

Algorithm
- Iterative
  - using forloop
  - do loop
  - do while loop.
- Recursive.

To find the time complexity of iterative algorithm, we have to count the no. of times the loop is executing.

For eg:- Consider the pgm segment

```
main()                                    To find the complexity.
{ int i,n;        → //O(1)  // declaration stmt.
  for(i=0; i<n; i++)  // loop gets executed n times.
      printf("Hello")
}
```

∴ Complexity is $O(n)$ //

Eg2.
```
A()
{ int i,j;
  for(i=1 to n) //→n times      // it
     for(j=1 to n) //→n times
        print("Hello")
}
```

∴ Time Complexity is $O(n^2)$ //

Eg3:-
```
A()
{ i=1, s=1
  while(s<=n)
     { i++
       s=s+i
       Print("Hello")
     }
```

Ans.

s	1	3	6	10	15	21	. . . n
i	1	2	3	4	5	6	

By the time we reach k iterations, $s > n$.

When $i=1$, s = Sum of first one natural no.

When $i=2$, s = " " " two natural nos

When $i=3$, s = Sum of " 3 natural no.

when $i=k$, s = Sum of first k natural no

ie $\frac{k(k+1)}{2}$

For loop to terminate = $\frac{k(k+1)}{2} > n$ ie $\frac{k^2+k}{2} > n$

$k = O(\sqrt{n})$ //

Eg 4.

```
A ()
{ int i, j, k, n.
  for (i = 1; i <= n; i++)
    { for (j = 1; j <= i; j++)
      { for (k = 1; k <= 100; k++)
        { printf ("Hello");
        }
      }
}
```

When i=1	i = 2	i = 3	i = 4	...	i = n
j = 1 time	j = 2 times	j = 3 times	j = 4 times		j = n times
k = 100 times	k = 2 x 100 times	k = 3 x 100 times	k = 4 x 100 times		k = n x 100 times

So total time = $100 + 2 \times 100 + 3 \times 100 + \cdots n \times 100$;

$= 100 \dfrac{n(n+1)}{2} = O(n^2)$

Eg:-5

```
A ()
{ int i, j, k, n;
  for (i = 1; i <= n; i++)
    { for (j = 1; j <= i^2; j++)
      { for (k = 1; k <= n/2; k++)
        { printf ("Hello");
        }
      }
}
```

i = 1	i = 2	i = 3
j = 1 time	j = 4 times	j = 9 times
k = n/2 x 1	k = n/2 x 4	k = n/2 x 9
.	.	.
i = n		
j = n^2		
k = n/2 x n^2		

Total time complexity = $n/2 \times 1 + n/2 \times 4 + n/2 \times 9 + \cdots + n/2 \times n^2$.

$= n/2 (1 + 4 + 9 + \cdots n^2)$ — Sum of squares of first n natural nos.

$= n/2 \left(\dfrac{n(n+1)(2n+1)}{6} \right)$

$= O(n^4)$.

Eg 6:

```
A()
{ for (i = 1; i <= n; i = i * 2)
    printf ("Hello")
}
```

i = 1, 2, 4 ... n

$2^0\ 2^1\ 2^2 \cdots 2^k$

$2^k = n$

$O(\log_2 n)$

Recurrence Relations :- Are used to determine the running time of recursive pgms.

A recursive algorithm looks like.

```
A(n)
{ if (..)
  { return (A(n/2) + A(n/2))        // There are two A(n/2) terms.
  }
}
```

To find the time complexity, we write a recursive ~~fn~~ function.
Let T(n) be be the time taken to solve A(n). For the if stmt to execute, it takes a constant time. Addition also const. time.

$\therefore T(n) = C + 2T(n/2)$. This is recursive fn or recurrence relation of above algorithm.

A recurrence relation is an equation that ~~define~~ represents a fn in terms of itself on smaller values of input.

There are 3 types of recurrence relation.

(1) **Divide & Conquer** :-

The general form is $T(n) = aT(n/b) + f(n)$ $a \geq 1$, $b > 1$

A problem of size n is divided into 'a' problems each having size n/b. & $f(n)$ is the cost of division.

eg:- $T(n) = 2T(n/2) + n$ — quick sort

$T(n) = T(n/2) + 1$ — Binary search.

These types of recurrence relations can be easily solved using Master Method.

(2) **Chip & Conquer**

The general form is $T(n) = T(n-k) + f(n)$. $k > 1$

(3) **Chip and be Conquer**

The general form is $T(n) = aT(n-k) + f(n)$ $a > 1$

There are different methods for solving recurrance relation.

(1) **~~Substitution Method~~ / Iteration method.**

Q Solve $T(n) = T(n/2) + 1$ $T(1) = 1$

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

~~put~~ Substitute $n = n/2$ $\therefore T(n/2) = T(n/4) + 1$

$\therefore T(n) = T(n/4) + 1 + 1 = T(n/4) + 2$

$T(n) = T(n/8) + 3$

$T(n) = T(n/16) + 4$.

Substituting k times.

put $n = 2^k$ $\therefore k = \log_2 n$

$T(n) = T(1) + \log n = \underline{O(\log n)}$.

$T(n/4) = T(n/8) + 1$

$T(n/8) = T(n/16) + 1$

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

in $T(n) = T(n/2^k) + k$.

(2) $T(n) = 2T(n/2) + n \qquad T(1) = 1$

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

$= 4T(n/4) + 2n$.

$\therefore T(n) = 4 \cdot [2T(n/8) + n/4] + 2n$

$= 8T(n/8) + 3n$

$T(n/8) = 16T(n/16) + 4n$.

$T(n/2) = 2T(n/4) + n/2$.

$T(n/4) = 2T(n/8) + n/4$

$T(n/8) = 2T(n/16) + n/8$.

Repeating up to k terms.

$T(n) = 2^k T(n/2^k) + kn$. , put $n = 2^k$ $\therefore k = \log n$

$T(n) = n T(1) + n \log n$

$\therefore T(n) = \underline{O(n \log n)}$.

(3) $T(n) = 2T(n/2) + 1 \qquad T(1) = 1$

$\therefore T(n) = 2 \cdot [2T(n/4) + 1] + 1 = 4T(n/4) + 3$

$\therefore T(n) = 4[2T(n/8) + 1] + 3$

$= 8T(n/8) + 7$.

$\therefore T(n) = 8[2T(n/16) + 1] + 1 = 16T(n/16) + 15$

$T(n/2) = 2XT(n/4) + 1$

$T(n/4) = 2T(n/8) + 1$

$T(n/8) = 2T(n/16) + 1$

Repeating upto k terms.

$T(n) = 2^k T(n/2^k) + 2^k - 1$ put $n = 2^k$

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

$= O(n)$.

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

$\therefore T(n) = 2 \cdot [2T(n/4) + \sqrt{n/2}] + \sqrt{n}$.

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

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

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

$T(n/4) = 2T(n/8) + \sqrt{n/4}$

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

$= n + n\sqrt{n} = O(n\sqrt{n})$.

$b^{\log_b x} = x$

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

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

Repeating upto k terms $2^k T\left(\frac{n}{2^k}\right) + \sqrt{n}\left[1 + \sqrt{2} + \sqrt{4} + \cdots + \sqrt{2^{k-1}}\right]$

$= 2^k T\left(\frac{n}{2^k}\right) + \sqrt{n}\left[1 + \sqrt{2} + 2\sqrt{2} + \cdots \text{. log n terms}\right)$

put $n = 2^k$ $\quad n T(1) + \sqrt{n}\left(\frac{1 \times (\sqrt{2})^{\log n} - 1}{\sqrt{2}-1}\right)$ $= n + n \cdot nT(1) + \sqrt{n} \times n$

$= O(n)$

(3) $T(n) = T(n-1) + 1 \quad T(1) = 1$

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

$\therefore T(n) = T(n-3) + 3$

$\therefore T(n) = T(n-4) + 1$.

Repeating upto k terms

$T(n) = T(n-k) + k$.

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

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

$= O(n)$.

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

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

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

$T(n-3) = T(n-4) + 1$

put $n = k+1 \quad \therefore k = n-1$

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

Substitute for $T(n/2)$ in $T(n)$

$T(n) = 2 \cdot [2T(n/4 + \log n - 1] + \log n$

$= 4T(n/4) + 3\log n - 2$.

$T(n) = 4 \cdot [2T(n/8) + \log n - 2] + 3\log n - 2$

$= 8T(n/8) + 7\log n - 10$

$T(n) = 8 \cdot [2T(n/16) + \log n - 3] + 7\log n - 10$

$= 16T(n/16] + 15\log n - 34$

Repealing upto k terms we have. & $n = 2^k$

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

$= 2^k T(1) + (n-1)\log n - C$

$= n T(1) + n \log n - \log n - C$

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

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

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

$= 2T(n/4) + \log n - 1$

$T(n/4) = 2T(n/8) + \log n/4$

$= 2T(n/8) + \log n - \log 4$

$= 2T(n/8) + \log n - 2$

$T(n/8) = 2T(n/16) + \log n/8$

$= 2T(n/16) + \log n - 3$

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

$\therefore T(n) = 2 \times [2T(n/4) + (n^2/2)] + n^2$

$= 4T(n/4) + n^2/2 + n^2$

$\therefore T(n) = 4 \times [2T(n/8) + (n/4)^2] + n^2/2 + n^2$

$= 8T(n/8) + \frac{n^2}{4} + n^2/2 + n^2$

$T(n/2) = 2T(n/4) + (n/2)^2$

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

Repealing upto k terms

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

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

$\therefore T(n) = O(n^2)$

Masters Theorem :- It is a direct way to get the soln of a recurrence relation, provided that it is of the fol- type.

$T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$ $a \geq 1$ and $b > 1$.

case 1 :-

Generic form : If $f(n) = \Theta(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$

Eg:- $T(n) = 8T(n/2) + 1000n^2$

Ans Here $a = 8$ $b = 2$ $f(n) = 1000n^2$ So

$f(n) = \Theta(n^c)$, where $c = 2$

$\log_b a = \log_2 8 = 3 > 2$

$\therefore T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$

Case2 :- If it is true for some constant $k \geq 0$ that $f(n) = \Theta(n^c \log^k n)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log^{k+1} n)$

eg:- $T(n) = 2T(n/2) + 10n$

Ans Here $a = 2$ $b = 2$ $f(n) = 10n$ $c = 1, k = 0$

$\log_b a = \log_2 2 = 1$. Here $c = \log_b a$

$\therefore T(n) = \Theta(n^1 \log^1 n) = \Theta(n \log n)$

Case3 :- If it is true that $f(n) = \Theta(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$

eg:- $T(n) = 2T(n/2) + n^2$.

Here $a = 2$ $b = 2$ $f(n) = n^2$ $c = 2$.

$\log_b a = \log_2 2^2 = 1$. Here $c > \log_b a$.

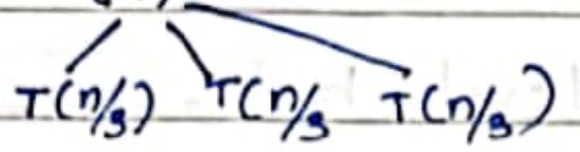
$\therefore T(n) = \Theta(f(n)) = \Theta(n^2)$

Solving Recurrences Using Recurrence Trees

Each node in the recursion tree has two fields, the size field and non recursive cost field. A node is represented as follows:

T(size)	nonrec. cost

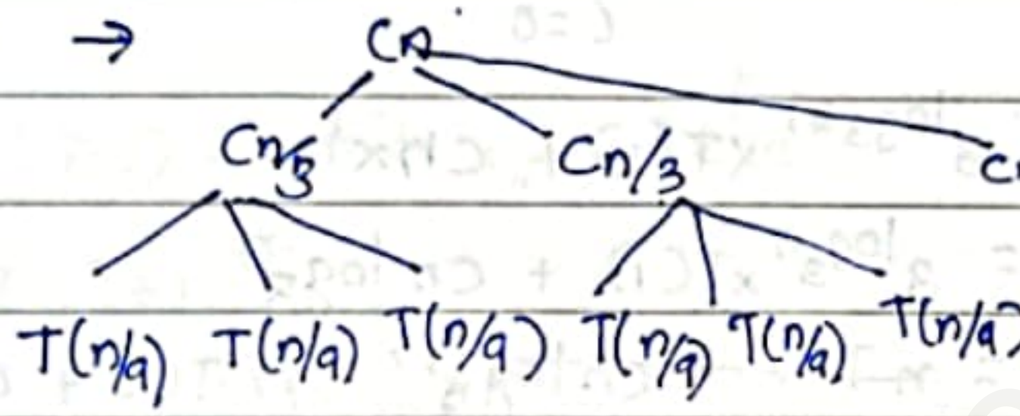
Eg $T(n) = 3T(n/3) + cn$ $\quad c > 1$

$T(n) \rightarrow$ 

First round.

The value of $T(n)$ is the sum of 4 nodes $[T(n/3), T(n/3), T(n/3), cn]$

Now Expand $T(n/3)$ to get one more level.

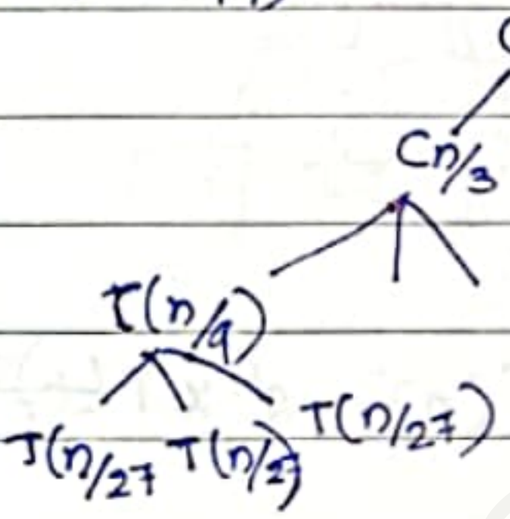
$T(n) \rightarrow$ 

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

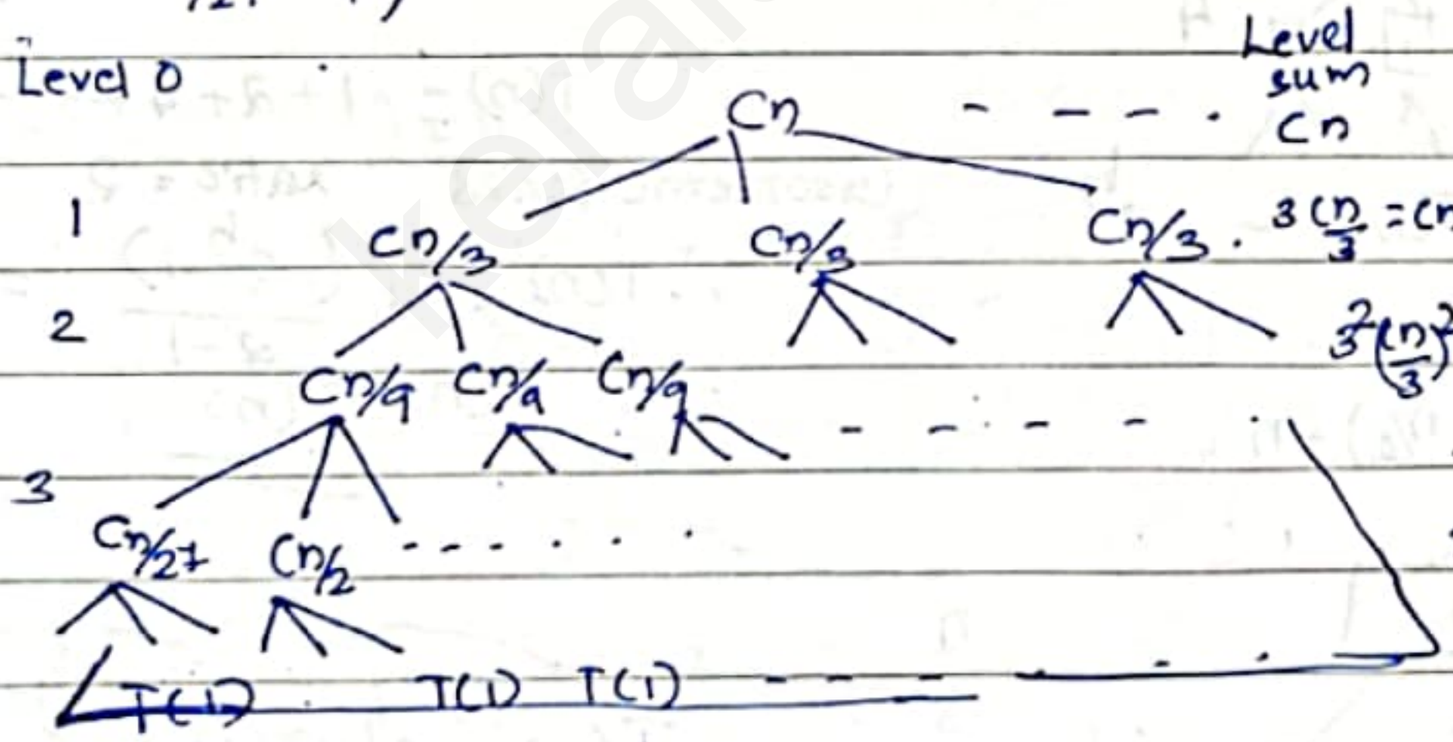
Second Round.

$T(n/9) = 3T(\frac{n}{27}) + c(n/9)$

Expand $T(n/9)$



Now we continue to expand T with th to smaller subproblem until we get subproblems of size 1. ie $T(1)$



At level 0 we have 1 node. At level 1, we have 3^1 nodes.

At level 2 we have 3^2 nodes. $\therefore$ At level i, we have 3^i nodes

If we call, the bottom most level as h, where h is the height of the tree, subproblems are of size 1 at this stage. Assuming $T(1)$ is a constant. No. of nodes at this level is 3^h.

To evaluate the value of $T(n)$, we sum up the costs at each node

and all the levels in the recursion tree. ie adding the costs of each level.

Level sum is same for each level. $= cn$

At level h, the size of the subproblem $= n/3^h$, h is the height of the tree which is assumed to be 1.

$\therefore n/3^h = 1$ ie $3^h = n$ $\quad h = \log_3 n$

$\therefore T(n)$ Upto level $h-1$, we have level sum $= cn$

Level h, level sum $= 3^h \times T(1)$

$$\therefore T(n) = 3^h \times T(1) + \sum_{i=0}^{h-1} cn.$$

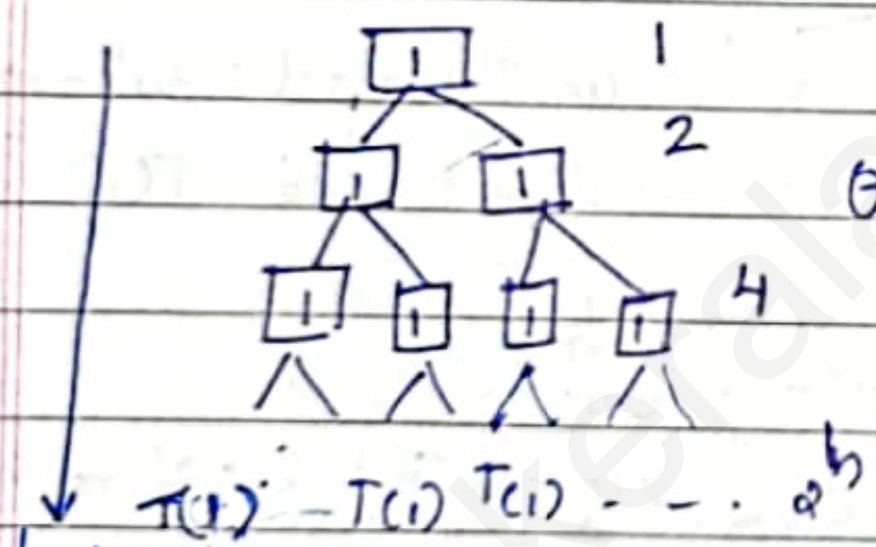
$$= 3^{\log_3 n} \times T(1) + cn \times h$$

$$= 3^{\log_3 n} \times T(1) + cn \log_3 n$$

$$= n T(1) + cn \log_3 n$$

$$= \underline{\Theta(n \log_3 n)}.$$

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



1
2
4
$2^h = n$
$h = \log n$

$\Theta(\log n)$

$T(n) = 2^h \times T(1) + \sum_{i=0}^{h-1} 2^i$

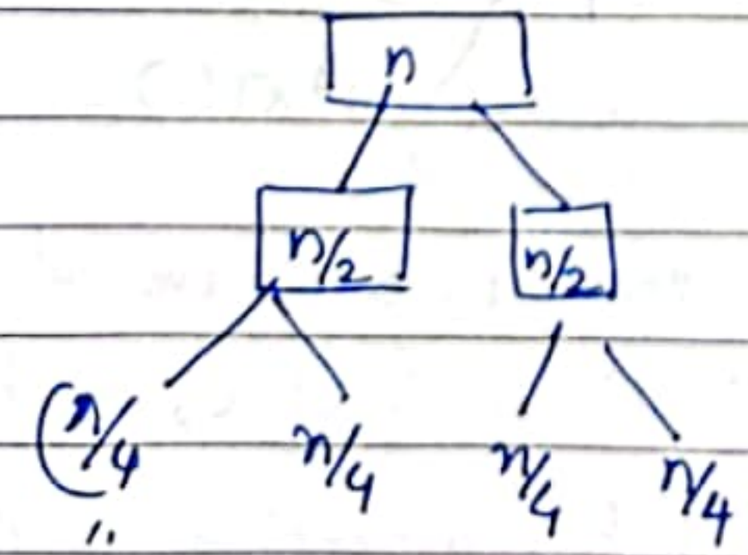
$T(n) = 1 + 2 + 4 + \ldots + 2^h$

Geometric series ratio = 2.

$\therefore T(n) = 1 \dfrac{(2^h - 1)}{2-1} = 2^{\log_2 n} - 1$

$T(n) = O(n)$

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



n
n
n
n

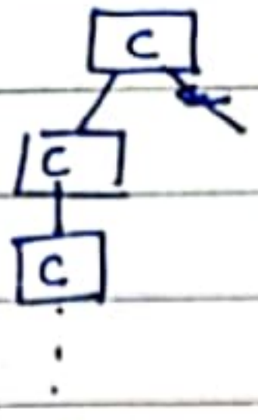
$$T(n) = 2^h \times T(1) + \sum_{i=0}^{h-1} n.$$

$$= 2^{\log_2 n} T(1) + nh.$$

$$= n \times T(1) + n \log_2 n$$

$$T(n) = \Theta(n \log_2 n)$$

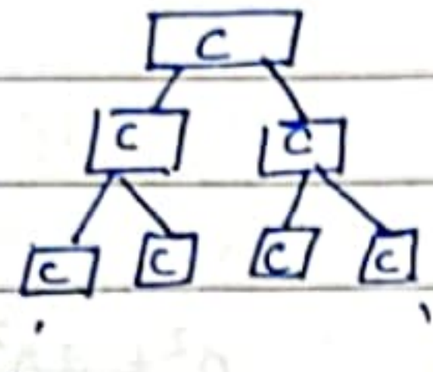
(4) $T(n) = T(n/2) + c$



$c + c + c + c$ - up to $\log_n$ terms.

$= O(\log n)$

(5) Solve $T(n) = 2T(n/2) + c$



c

2c

4c

$2^h c$

$c + 2c + 4c + \cdots + 2^h c$

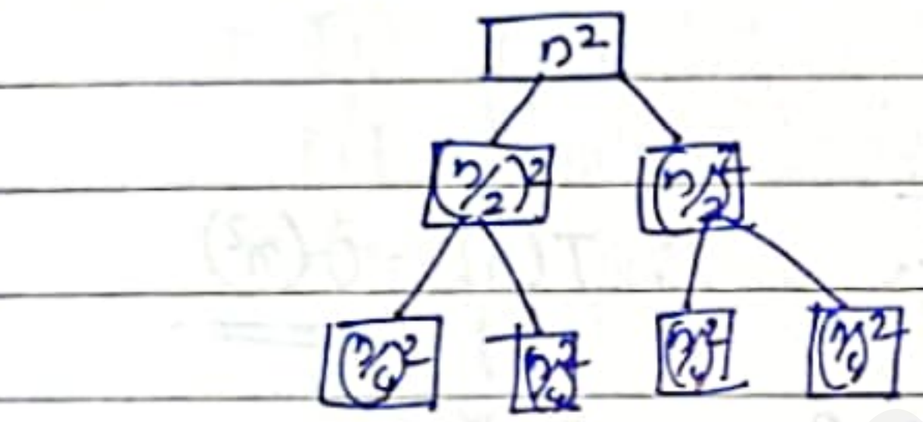
$1 + 2^1 c + 2^2 c + \cdots + 2^h c$

$= \frac{1(2^h - 1)}{2-1} = \frac{2^{\log_2 n} - 1}{1}$

$S = \frac{a(r^n - 1)}{r-1}$

height h - $T(1) - T(1) - \cdots$ $\therefore T(n) = O(n)$

(6) $T(n) = 2T(n/2) + n^2$.



n^2

$n^2/2$

$n^2/4$

$n^2/2^h$

$n^2 + n^2/2 + n^2/4 + \cdots + n^2/2^h$

Geometric series with common ratio $r = 1/2$

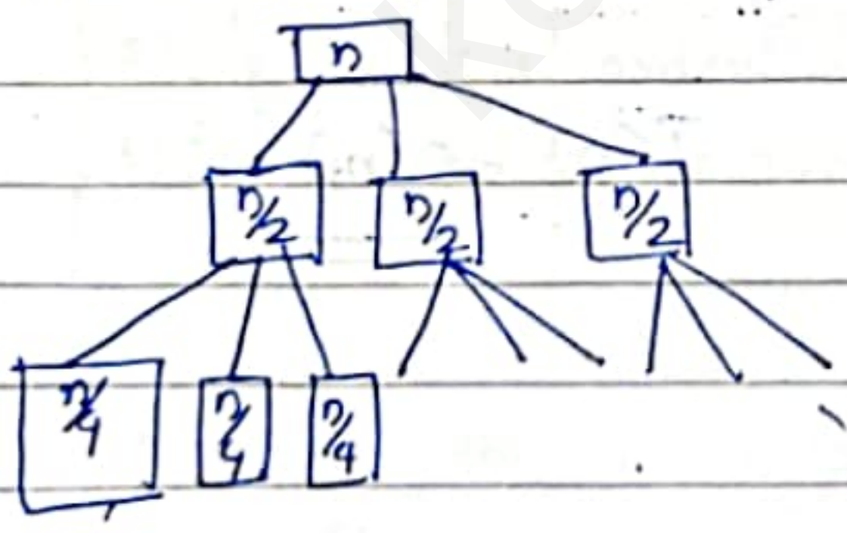
$T(n) = n^2 \left[\frac{1 - 1/2^h}{1/2}\right]$

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

$= 2n^2 - n$

$\therefore T(n) = O(n^2)$

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



n

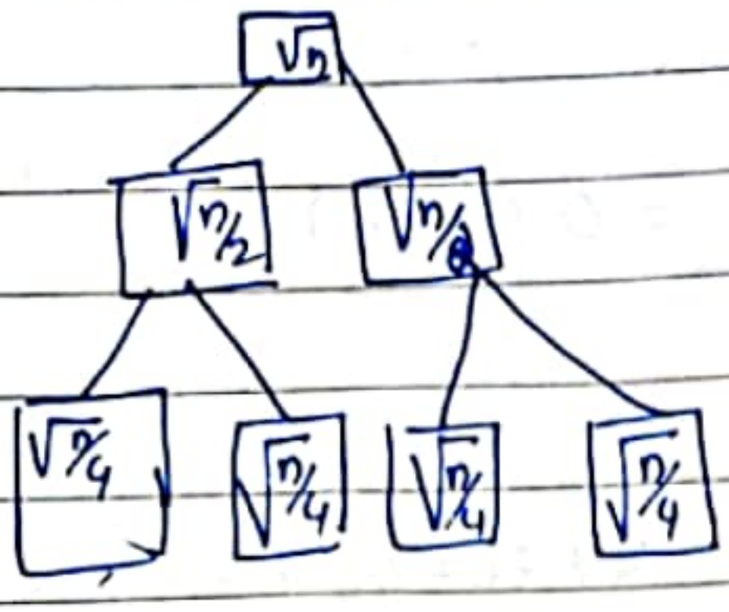
$3n/2$

$9n/4$

$3^h \times n/2^h$.

$O(n^2)$

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



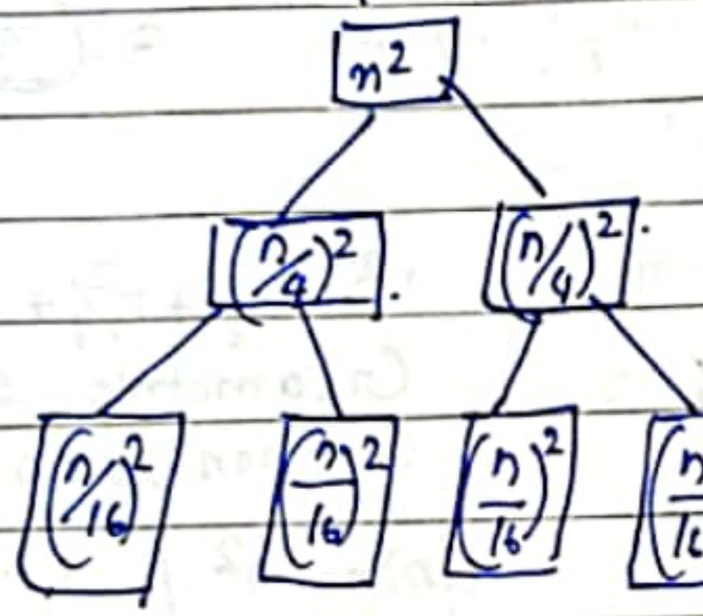
$\sqrt{n}$

$\sqrt{2n}$

$2\sqrt{n}$.

$O(n\sqrt{n})$

(9) $T(n) = 2T(n/4) + n^2$.



n^2.

$\frac{2n^2}{4^2}$

$\frac{4n^2}{16^2}$

$\vdots$

$\vdots$

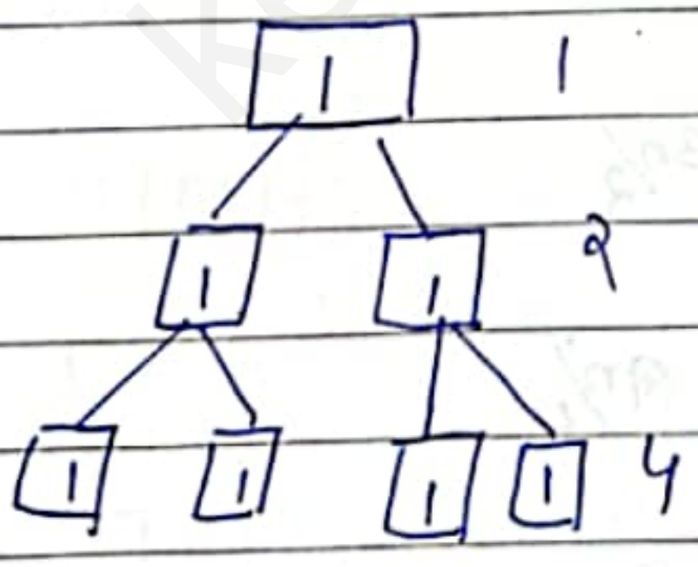
$\frac{2^h \times n^2}{4^h}$.

$n^2 + \frac{2n^2}{4^2} + \frac{4n^2}{16} + \cdots + \frac{2^b \times n^2}{4^b}$

Geometric series with ratio $= 1/2^3$

$\therefore T(n) = O(n^2)$.

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



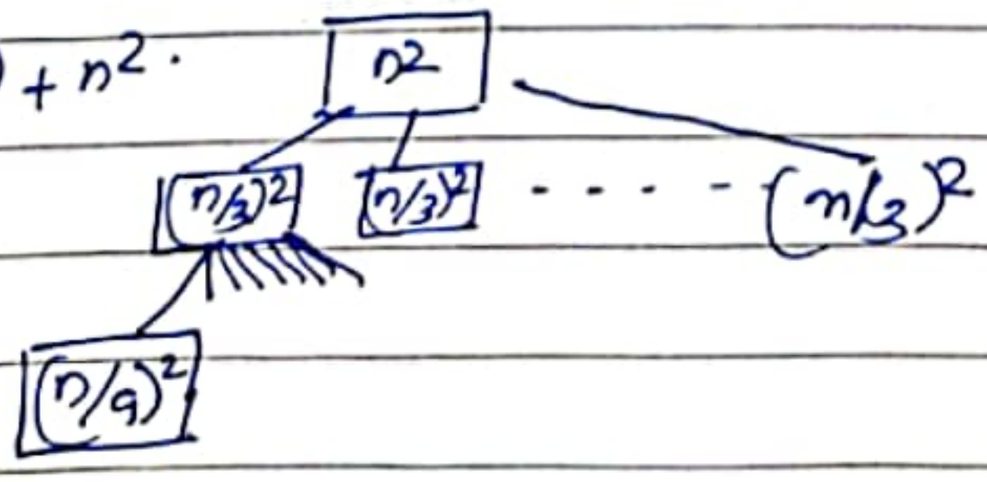
1

2

4

$T(n) = O(n)$

(11) $9T(n/3) + n^2$.



n^2

$9 \times (n/3)^2 = n^2$

$81 \times (n/9)^2 = n^2$

$T(1)$

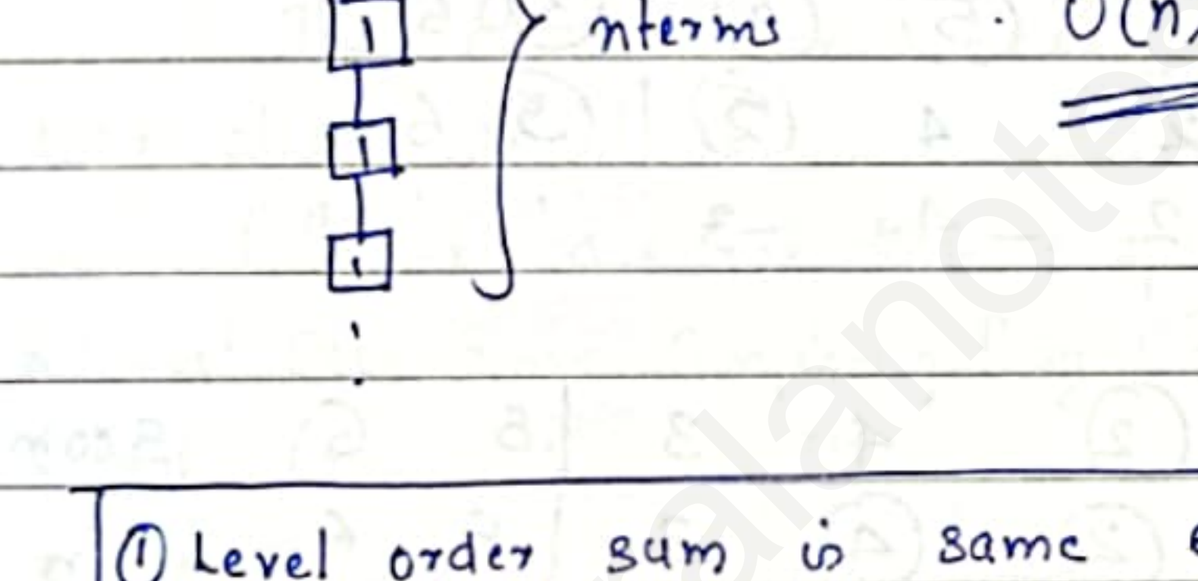
$$T(n) = q^h \times T(1) + \sum_{i=0}^{h-1} n^2 \qquad h = \log_q n$$

$$= q^h \times T(1) + n^2 h$$

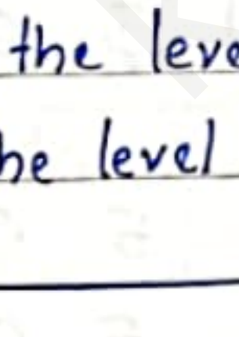
$$= q^{\log_q n} \times T(1) + n^2 \log_q n \quad = \quad n T(1) + n^2 \log_q n .$$

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

(12) $T(n) = 3T(n/2) + \sqrt{n}$



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



$1 + 1 + \cdots + n$ terms

$O(n)$

① Level order sum is same eg (1, 1, 1 ...) (n, n, n - .)
then $T(n) = O(\log n \times \text{first term.})$

(2) If the level order sum is a G.P with $r < 1$ then $T(n) = O(\text{first term})$

(3) If the level order sum is a G.P with $r > 1$ then $T(n) = O(n \times \text{first term})$

Estimating the Space & time complexity of bubble sort

Bubble sort is a simple alg which is used to sort a given set of n elements in an array. The algorithm proceeds by comparing the first element of the array with the second element, if the first element is greater than second element, it will swap both the elements, and then move on to compare the second & the third element, & soon

If we have n elements, then we need to repeat this process for n-1 times.

It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up toward the last place or the highest index.

eg:- (5) (1) 6 2 4 3

5>1 interchange 1 (5) (6) 2 4 3

5<6 No swapping 1 5 (6) (2) 4 3

6>2. Swap 1 5 2 (6) (4) 3

6>4 Swap 1 5 2 4 (6) (3)

6>3 Swap 1 5 2 4 3 | 6

} 5 comparisons in pass 1

Pass 2.

1<5 No swap (1) (5) 2 4 3 6

5>2 Swap 1 (5) (2) 4 3 6

2<5 No swap 1 (2) (5) (4) 3 6

5>4 Swap 1 2 (5) (4) (3) 6

5>3 Swap 1 2 4 (5) | (5) 6

1 2 4 3

} 4 comparisons in pass 2

pass 3.

1<2 No swap (1) (2) 4 3 | 5 6

2<4 No swap 1 (2) (4) 3 | 5 6

4>3 Swap 1 2 (4) (3) | 5 6

1 2 3 4 | 5 6

3 comparisons in pass 3.

pass 4

(1) (2) 3 | 4 5 6

1 (2) (3) | 4 5 6

2 comparisons in pass 4.

pass 5

(1) (2) | 3 4 5 6

1 Comparison in pass 5.

Algorithm for bubble Sort.

```
Bubble-sort ( A, n )
{
    for(i=0; i<n; i++)
        {
            for(j=0; j<n-i-1; j++)
            { if (A[j] > A[j+1])
```

```
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}
```

Time Complexity Analysis of Bubble sort

In Bubble sort, n-1 comparisons will be done in the I^st pass, n-2 in 2^nd pass, n-3 in 3^rd pass and so on. So. total no. of comparisons

$= (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1$

$= \frac{n(n-1)}{2}$. ie $O(n^2)$.

Space Complexity of Bubble sort

Space complexity for Bubble sort is O(1). because only a single additional memory space is reqd, ie for temp variable.

The best case time complexity will be O(n), it is when the list is already sorted.

Following are the time & space complexity for the Bubble sort alg.

* Worst Case Time Complexity [~~Big-O~~] = $O(n^2)$.
* Best Case " " = $\Omega(n)$
* Averge Case " " = $\Theta(n^2)$
* Space Complexity = $O(1)$ //.

Amortized Complexity

~~of~~ In an amortized analysis, the time reqd to perform a sequence of datastructure operns is avg over all operns performed. Amortized analysis can be used to show that avg cost of an opern is small. Unlike the avg probability distribution fn, the amortized analysis gaurantees that the avg performance of each opern in the worst case.

Mainly three techniques are used in amortized analysis. The main diff. is the way the cost is assigned.

1. Aggregate method :-

* It computes the worst case time T(n) for a sequence of operns.
* The amortized cost is T(n)/n per opern.
* It gives the avg performance of each opern in the worst case.

Calculate the time complexity of fol. functions.

```
main()
{  for (i=1; i<=n; i*2)
            sum = sum + i + fun(i);
}
void func (int m)
 {
   for j=1; j<=m; j++)
      statement with o(1) complexity
   }
```

Soln → for loop in main() will successfully execute in $\log_2 n$ times

→ for loop in fun() with execute $2^0, 2^1, 2^2, \ldots\ldots 2^{\log n}$ time

$$\text{Complexity} = 2^0 + 2^1 + 2^2 + \ldots\ldots 2^{\log n}$$

$$= 1\ \frac{(2^{\log n+1} - 1)}{2-1}$$

$$= 2 \times 2^{\log n} - 1$$

$$= 2 \cdot n^{\log 2} - 1$$

$$= 2n - 1 = O(n)$$

Sum of n terms in a G.P = $\frac{a(r^n-1)}{r-1}$

Asymptotic Notations Defn

$O \rightarrow 0 \leq f(n) \leq cg(n)$

$\Omega \rightarrow f(n) \geq cg(n) \geq 0$

$\Theta \rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

$o \rightarrow 0 \leq f(n) < cg(n)$

$\omega \rightarrow f(n) > cg(n) \geq 0$

Find the O notation of the gol. fn.

(a) $f(n) = 3n+2$.

Soln Soln :- Definition of O $\Rightarrow$ $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

Here $f(n) = 3n+2$

We take $g(n)$ as the highest power of n.

$\therefore g(n) = n$

We take c as the co-effecient of the highest power of n plus one. or we can choose a random value.

$\therefore c = 3+1 = 4$

$\therefore 3n+2 \leq 4n$, for all $n \geq n_0$.

Calculalé n_0.

put $n=1 \Rightarrow$	LHS = 5	RHS = 4	False
$n=2 \Rightarrow$	LHS = 8	RHS = 8	True
$n=3 \Rightarrow$	LHS = 11	RHS = 12	True
$n=4 \Rightarrow$	LHS = 14	RHS = 16	True.

The above eqn is lrue when $n \geq 2$. $\therefore n_0 = 2$.

$\therefore 3n+2 \leq 4n$ for all $n \geq 2$.

$\therefore f(n) = O\,g(n)$

$\therefore 3n+2 = O(n)$

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

Soln. Defn of O $\Rightarrow$ ~~$4n^3 + 2n + 3$~~ $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

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

if $n=1$	LHS = 9	RHS = 5	False
$n=2$	LHS = 39	RHS = 40	True.
$n=3$	LHS = 117	RHS = 135	True . .

The above eqn is lrue when $n \geq 2$ $\therefore n_0 = 2$.

$\therefore 4n^3 + 2n + 3 \leq 5n^3$ for all $n \geq 2$.

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

$\therefore 4n^3 + 2n + 3 = O(n^3)$

(c) $f(n) = 2^{n+1}$

Soln $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

Here $f(n) = 2^{n+1}$ $g(n) = 2^n$ $c = 2$

$\therefore 2^{n+1} \leq 2\,2^n$ for all $n \geq n_0$

If $n=1$ LHS $= 4$ RHS $= 4$ True.

$n=2$ LHS $= 8$ RHS $= 8$ True.

$n=3$ LHS $= 16$ RHS $= 16$ True.

$\therefore$ The above eqn is true when $n \geq 1$

$\therefore 2^{n+1} \leq 2\,2^{2n}$ for all $n \geq 1$

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

$2^{n+1} = O(2^n)$ //

(d) Is $2^{2n} = O(2^n)$?

Analysis of Recursive Algorithms

Steps to analyse recursive algorithms

1. Identifying input size of smaller subproblems.
2. Writing recurrence relation for the time complexity.
3. Solving recurrence relation to get the time complexity.

In the case of recursion, we solve the problem using the soln. of the smaller subproblems.

→ If the time complexity fn of the alg. is $T(n)$ then the time complexity of the smaller subproblems will be defined by the same fn, but in terms of i/p size of the sub problems.

→ Suppose we have K number of sub problems.

$\therefore T(n) = T$(i/p size of subproblem 1) + T(i/p size of sub problem 2) + ... + T(i/p size of subproblem K) + Time complexity of additional operations other than recursive calls.

The above equation is called the recurrence relation.

→ A recurrence relation is an equation that describes a sequence where any term is defined in terms of its previous terms.

→ We use recurrence relation to define and analyze time complexity of recursive algorithms in terms of i/p size.

→ After writing the recurrence relation for a recursive algorithm, we solve it & calculate the overall time complexity in terms of Big-Oh Notation.

eg:- Binarysearch (A[ ], low, high, key)

```
{
    if (low > high)
        return -1;
    if (  mid = low + high / 2 .
    if ( A[mid] == key)
        return mid+1;
    else if (A[mid > key)
        BinarySearch ( A[ ], low, mid-1, key )
    else
        Binarysearch (A[ ], mid+1, high, key)
}
```

→ At each step of recursion, we are doing one comparison & decreasing the i/p size by half

→ In other words, we are solving the problem of 'n' size by the soln of one sub problem of ~~siz~~, i/p size 'n/2'

→ Time complexity T(n) = Time Complexity of n/2 size problem + Time complexity of comparison oper.

∴ $T(n) = T(n/2) + O(1)$

∴ Recurrence Relation of binary search

$T(n) = T(n/2) + C$, where $T(1) = C$

Methods to solve recurrence Relation.

1) Iteration Method
2) Recursion Tree Method (Recurrence Tree Method)
3) Substitution Method.
4) Master's Theorem.

(3) Substitution Method.

The substitution method for solving recurrences characterize comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants & show that the soln works.

This method is powerful, but we must be able to guess ~~the~~ the form of the answer in order to apply it.

eg:- $T(n) = 2T(n/2) + kn \quad T(1) = 1$

Soln. Guess the soln for $T(n) = O(n \log n)$.

According to defn of O; $T(n) = O(n \log n)$ if $T(n) \leq c\, n \log n$, there exists $c > 0$ such that $\forall\, n \geq n_0$ the above condn holds

~~Assume that $T(n) \leq O(n \log n)$ is true for all $m < n$~~

Assume that it is true for $m < n$.

We need to Prove: $T(n) \leq cn \log n$ assuming $T(m) \leq cm \log m \quad \forall\, m < n$.

Let $m = n/2 < n$.

$\therefore$

$\therefore T(n/2) \leq c\, n/2 \log n/2$

We have $T(n) = 2T(n/2) + kn \rightarrow (1)$

Substitute for $T(n/2)$ in (1)

$T(n) \leq c\, n/2 \log n/2 + n$

$= cn \log n/2 + n. = cn\{\log n - \log 2^2\} + kn$

$= cn(\log n - 1) + n \quad = cn \log n - cn + kn$

$\leq cn \log n \quad = cn \log n - (k - c)n$

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

Mistakes in Substitution method.

eg:- $T(n) = 2T(n/2) + kn$.

Soln Guess $T(n) = O(n)$ then $T(n) \leq kn$

Assume: $T(m) = O(m) \leq cm \quad \forall\ m < n$.

To prove $T(n) \leq cn$ assuming $T(m) \leq cm \quad \forall\ m < n$.

$T(n) = 2T(n/2) + kn \rightarrow (1)$

Assume $m = n/2 < n$.

$T(n/2) \leq cn/2$. put it in (1)

$T(n) \leq 2cn/2 + kn$.

$= cn + kn$.

$T(n) \leq cn + kn$

But we want to get $T(n) \leq cn$ ∴ But ~~we~~ got

The mathematical induction fails.

∴ $T(n) \neq O(n)$.

Disadvantage of substitution method:- It is very prone to errors if the guess for soln is not correct.

(2) $T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) + n & n > 1 \end{cases}$

Soln Guess $T(n) = O(n^2)$ then $T(n) \leq cn^2$ for some const c & $n_0 \leq n$.

∴

Assume $T(m) = O(m^2) \leq cm^2 \quad \forall\ m < n$.

To prove $T(n) \leq cn^2$ assuming $T(m) \leq cm^2 \ \forall\ m < n$

$T(n) = T(n-1) + n \rightarrow (1)$

put $m = n-1$ ∴ $T(n-1) \leq c(n-1)^2$.

~~put~~ Substitute for $T(n-1)$ in (1).

∴ $T(n) \leq c(n-1)^2 + n$

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

$= cn^2 - 2cn + c + n$

$= cn^2 - (cn + c - n)$

$\leq cn^2$

$\therefore$ By mathematical induction we prove that

$T(n) \leq cn^2$.

$\therefore T(n) = O(n^2)$

(3) $T(n) = \begin{cases} 1 & n=1 \\ 2T(n/2)+1 & n>1 \end{cases}$

soln Guess $T(n) = O(n)$ $\therefore T(n) \leq cn$ for $c>0, n \geq n_0$.

Assume $T(m) = O(m)$

$\therefore T(m) \leq O(m)$ $\forall m < n$.

Let $m = n/2 < n$.

We have $T(n) = 2T(n/2)+1 \rightarrow (1)$

$\therefore T(m/2) \leq 2 \cdot T$

$\therefore T(n/2) \leq cn/2$

Substitute for $T(n/2)$ in (1)

$\therefore T(n) \leq 2[cn] + 1 = cn + 1 \not\leq cn$

which doesnot imply that $T(n) \leq cn$

But $T(n) = O(n)$

Let's try $T(n) \leq cn - b$ where b is a const.

$\therefore$ Assume $T(m) = O(m)$

$\therefore T(m) \leq O(m)$ $\forall m < n$.

Let $m = n/2 < n$.

We have $T(n) = 2T(n/2)+1$

$T(n/2) \leq cn/2 - b$.

$\therefore$ (1) becomes.

$T(n) \leq 2\left[cn/2 - b\right] + 1$

$= cn - 2b + 1$ if $-b+1 \leq 0$

$\leq cn - b$ $b \geq 1$

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

Masters Theorem

→ Master's Method is a direct way to get the soln.
→ The master's method works only for the fol. type of recurrences or for recurrences that can be transformed to fol. type.

$T(n) = aT(n/b) + f(n)$

where $a \geq 1, b > 1$ & $f(n) > 0$.

For eg:- $T(n) = T(n/2) + c$

Here $a = 1$, $b = 2$ & $f(n) = c$.

eg2:- $T(n) = 2T(n/2) + n$

Here $a = 2$, $b = 2$ $f(n) = n$.

Master's Theorem

Taking an equation of the form

$T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$ & $f(n) > 0$

The Master's Theorem states:

* case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$
then $T(n) = \Theta(n^{\log_b a})$
* case 2: if $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
* case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ &
if $af(n/b) \leq cf(n)$ for some $c < 1$ & all sufficiently large n, then $T(n) = \Theta(f(n))$

eg:- $T(n) = 2T(n/2) + n$

Soln Here $a = 2$ $b = 2$ $f(n) = n$.

$n^{\log_b a} = n^{\log_2 2} = n$.

Comparing $n^{\log_b a}$ with $f(n) \Rightarrow f(n) = \Theta(n^{\log_b a})$

∴ Case 2 can be appled. $T(n) = \Theta(n^{\log_b a} \log n)$

∴ $T(n) = \Theta(n \log n)$

(b) $T(n) = 2T(n/2) + n^2$.

Here $a = 2$, $b = 2$ $\quad f(n) = n^2$

$n^{\log_b a} = n^{\log 2^2} = n$ $\quad \therefore$ comparing $f(n)$ with $n^{\log_b a}$

$f(n) = n^2 \Rightarrow f(n) = \Omega(n^{1+\varepsilon})$ $\quad \varepsilon = 1$

case 3 can be applied

The condn is $af(n/b) \leq cf(n)$

$af(n/b) = 2f(n/2) = 2n^2/4 = n^2/2 \leq (1/2)n^2$

(for $c = 1/2$)

So the condn is satisfied for $c = 1/2$.

Thus $T(n) = \Theta(f(n)) = \Theta(n^2)$ //

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

Here $a = 2$, $b = 2$ $\quad f(n) = \sqrt{n}$

$n^{\log_b a} = n^{\log_2 2} = n$

$f(n) = O(n^{1-\varepsilon})$ case 1

$\therefore T(n) = \Theta(n)$

(d) $T(n) = 3T(n/4) + n\log n$.

Here $a = 3$ $\quad b = 4$ $\quad f(n) = n\log n$.

$n^{\log_b a} = n^{\log_4 3} = \Theta\ n^{0.792}$

$f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ case 3.

$af(n/b) \leq cf(n)$ $\quad$ i.e. $af(n/b) = 3(n/4)\log n/4 \leq 3/4\, n\log n = c \times f(n)$, $c = 3/4$

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

Case-1 $f(n) < n^{\log_b a} \Rightarrow$ $T(n) = \Theta(n^{\log_b a})$

Case-1 $f(n) < n^{\log_b a}$ $\quad T(n) = \Theta(n^{\log_b a})$

case2 $f(n) = n^{\log_b a}$ $\quad T(n) = \Theta(n^{\log_b a} \log n)$

case3 $f(n) > n^{\log_b a}$ $\quad T(n) = \Theta(f(n))$.

Properties of ~~Asymptotic~~ Notations

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

$a = 9 \quad b = 3 \quad f(n) = n$.

$n^{\log_b a} = n^{\log_3 9} = n^2$

$f(n) < n^2$.

$\therefore T(n) = \Theta(n^2)$ //

$f(n) < n^{\log_b a}$

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

$a = 1 \quad b = 3/2 \quad f(n) = 1$

$n^{\log_b a} = n^{\log 1/3/2} = n^0 = 1$

case 2 $f(n) = n^{\log_b a}$.

$\therefore T(n) = \Theta(n^{\log_b a} \log n)$

$= \Theta(\log n)$.

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

$a = 2 \quad b = 2 \quad f(n) = n \log n$.

$n^{\log_b a} = n^{\log 2^2} = n$.

~~$f(n) = n < n$~~ $\quad n \log n > n$

Here $f(n) = \Omega(n^{\log_b a + \varepsilon})$

$= a f(n/b) = 2 \times n/2 \log n/2$

$n \log n/2 \leq c n \log n$.

Here $c = 1$.

So case 3 cannot be applied.

Solve it using other method.

(h) $T(n) = 2T(n/2) + \Theta(n)$

$a = 2 \quad b = 2 \quad f(n) = \Theta(n)$.

$n^{\log_b a} = n$

case 2 applies

$\therefore T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$

(i) $T(n) = 8T(n/2) + \Theta(n^2)$

$a = 8 \quad b = 2 \quad f(n) = \Theta(n^2)$.

$n^{\log_b a} = n^{\log_2 8} = n^3$

$n^2 < n^3 \Rightarrow f(n) = O(n^{3-\varepsilon}), \varepsilon = 1$

case 1 applies.

$T(n) = \Theta(n^{\log_b a}) = \underline{\Theta(n^3)}$.

(j) $T(n) = 7T(n/2) + \Theta(n^2)$

$a = 7 \quad b = 2. \quad f(n) = \Theta(n^2)$

$n^{\log_b a} = n^{\log_2 7}$ $\quad 2.80 < \log_2 7 < 2.81$

$f(n) = O(n^{\log_2 7 - \varepsilon})$ case 3.

$\therefore T(n) = \Theta(n^{\log_2 7})$.

Properties of Asymptotic Notations.

Many of the relational properties of real numbers apply to asymptotic comparisons also.

(a) Transitivily

$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$

$f(n) = O(g(n)) \;\&\; g(n) = O(h(n))$ imply $f(n) = O(h(n))$

$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$

$f(n) = o(g(n)) \;\&\; g(n) = o(h(n))$ imply $f(n) = o(h(n))$

$f(n) = \omega(g(n)) \;\&\; g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$

(b) Reflexivity

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

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

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

(c) Symmetry

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

(a) 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 $g(n) = \omega(f(n))$.

Because these properties holds for asymptotic notations we can drawn an analogy bln the asymptotic comparison of two functions f & g and the comparison of two real numbers a & b.

$f(n) = O(g(n))$ is like $a \leq b$

$f(n) = \Omega(g(n))$ is like $a \geq b$.

$f(n) = \Theta(g(n))$ is like $a = b$

$f(n) = o(g(n))$ is like $a < b$

$f(n) = \omega(g(n))$ is like $a > b$.

$f(n)$ is is asymptotically smaller than $g(n)$ if $f(n) = o(g$
& $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g($

One property of real nos, however doesnot carry over to asymptotic notation:

Trichotomy :- For any two real nos. a & b, exactly one of the fol. must hold $a < b$, $a = b$ or $a > b$.

Although any two real nos can be compared, not all functions are asymptotically comparable. ie for two fns, $f(n)$ & $g(n)$, it may be case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For eg:- we cannot compare the fns n & $n^{\sin n}$ using asymptotic notations since the value of $n^{\sin n}$ oscillates bln 0 & 2, taking on all values in between.