1

```
MergeSort(A,l,r){
    if (l<r){
        m = ⌈(l+r)/2⌉

        Mergesort(A,l,m);
        Mergesort(A,m+1,r);
        Merge~~sort~~(A,l,m+1,r);
    }
}

Merge(A,l,m,r){
    x=l;
    y=m;
    k=l;
    while (x<m && y<=r){
        if (A[x]<A[y])
        {
            B[k]=A[x];
            x++;
            k++;
        }
        else{
            B[k]=A[y];
            k++;
            y++;
        }
    }
```

```
while(x<m)
{
    B[k]=A[x];
    k++;
    x++;
}
while(y<=r){
    B[k]=A[y];
    k++;
    y++;
}
for(i=l; i<=r; i++)
{
    A[i]=B[i];
}
}
```

Time Complexity

$$\begin{cases} T(n) = 2T(n/2) + O(n) & \text{when } n>1 \\ \quad\;\; = 1 & \text{when } n=1 \end{cases}$$

Solving this recurrence relation,

$T(n) = O(n \log n)$ [See in chapter 1],,

```
QuickSort (A, l, r){
    if (l < r){
        p = partition(A, l, r);
        QuickSort (A, l, p-1);
        QuickSort (A, p+1, r);
    }
}

partition(A, l, r){
    x = l;
    y = r;
    pivot = A[l];
    while (x<y){
        while (A[x] ≤ pivot)
            x++;
        while (A[y] > pivot)
            y--;
        if (x<y){
            t = A[x];
            A[x] = A[y];
            A[y] = t;
        }
    }
    A[l] = A[y];
    A[y] = pivot;
    return y;
}
```

Best Case Analysis

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

$= 1$ when $n = 1$

Solving for recurrence relation
$T(n) = \Theta(n \log_2 n)$ [see in chapter 1]

Case between worst and best

$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$ if $n > 1$

$= 1$ if $n = 1$

Solving this recurrence relation

$T(n) = \Theta(n \log_{10/9} n)$ [see in chapter 1 (tree)]

Worst Case Analysis

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

$= 1$ when $n = 1$

3

Master Method

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

$aT\left(\frac{n}{b}\right)$

→ ~~Case I~~:

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

~~$n^{\log_b a} = n.(\log_2 4)$~~

$n^{\log_b a} = n^{\log_2 4} = n^2$

Case I

$f(n) \neq O(n^{\log_b a - \varepsilon})$ for some ε,

because,

$n^2 \log n \,?\, n^{2-\varepsilon}$

$n^2 \log n \,?\, n^2 . n^{-\varepsilon}$

$n^2 \log n \,?\, \frac{n^2}{n^{\varepsilon}}$

~~Since, $n^2 \log n$ is asymptotically greater than n^2, case I failed.~~

clearly, case I failed.

Case II

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

i.e. $n^2 \log n \neq \Theta(n^2)$

So, case II failed.

case III

$f(n) \neq \Omega(n^{\log_b a + \varepsilon})$ for some ε,

because,

$\cancel{n^2 \log n \neq \Omega(n^2 + \varepsilon)}$

$n^2 \log n \ ?\ n^{2+\varepsilon}$

$n^2 \log n \ ?\ n^2 \cdot n^{\varepsilon}$

Since, n^{ε} is asymptotically greater than $\log n$,
case III failed.

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

$T\left(\frac{n}{b}\right)$

$f(n) = n \log n$

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

case I :

$f(n) \neq O(n^{\log_b a - \varepsilon})$

because,

$n \log n \ ?\ n^{1-\varepsilon}$

$n \log n \ ?\ \frac{n}{n^{\varepsilon}}$

clearly, case I failed.

case II

$f(n) \neq \Theta(n^{\log_b a})$

or, $n \log n \neq \Theta(n)$

so, case II failed.

Case III

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

because;

$n \log n \overset{?}{=} n^{1+\varepsilon}$

$n . \log n \overset{?}{=} n^1 . n^\varepsilon$

Since, n^ε is asymptotically greater than $\log n$, case III failed

(b)

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

~~$f(n) = n^{\log_2 2} = n$~~

$f(n) = n$

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

case I

$f(n) \neq O(n^{\log_b a - \varepsilon})$

because,

$n \overset{?}{=} n^{1-\varepsilon}$

$n \overset{?}{=} \frac{n}{n^\varepsilon}$

~~Since~~, ~~Case~~ :

clearly, case I failed.

Case II

$f(n) = n$

$n^{\log_b a} = n$

Here,

~~$f(n) \overset{?}{=} n^{\log_b a}$~~

∴ As $f(n) = n^{\log_b a}$, ∴ $T(n) = \Theta(n \log n)$

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

~~Case I:~~

$f(n) = n$

$n^{\log_b a} = n^{\log_2 4} = n^2$

Case I :

$f(n) \neq O(n^{\log_b a - \varepsilon})$ for some constant ε,

because,

$n \ ? \ n^{2-\varepsilon}$

$n \ ? \ \frac{n^2}{n^\varepsilon}$

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

$= \Theta(n^2)$

4 Master method

$T(n) = 16\,T\left(\frac{n}{4}\right) + n$

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$f(n) = O(n^{\log_b^a - \epsilon})$ for some $\epsilon < 0$

Given

$a = 16$

$b = 4$

Case I $\quad n^{\log_b a} = n^{\log_4 16} = n^2$

$\therefore f(n) = O(n^{2-\epsilon})$ for $\epsilon \leq 1$

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

$= O(n^2)$

Case-II

$f(n) \neq \Theta(n^{\log_b a})$

$n \neq n^2$

case failed

Case 3

$f(n) = \Theta(n^{\log_b a + \epsilon})$

$= O(n^{2+\epsilon})$

case failed

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

Given

$a = 7$

$b = 2$

$f(n) = n^2$

now,

case - I $\quad n^{\log_b a} = n^{\log_2 7} = n^{2.8}$

$\therefore f(n) = O(n^{2.8 - \epsilon})$ for $\epsilon \leq 0.8$

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

$= \Theta(n^{2.8})$

case - II

$f(n) \neq \Theta(n^{\log_b a})$

$n^2 \neq n^{2.8}$

case failed #

case III

$f(n) = \Theta(n^{\log_b a + \epsilon})$

$= O(n^{2.8 + \epsilon})$

case failed

5 # Fibonacci series

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

$= 1$ when $n = 1$ or $n = 2$

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

To be proved : $T(n) \leq c\, 2^n$

Proof:

Basis Case : When $n = 1$

$T(1) \leq c.2^1$

$1 \leq 2c \quad \forall\, c \geq 1$

It is true

When $n = 2$

$T(2) \leq c.2^2$

$1 \leq 4c \quad \forall\, c \geq 1$

It is trivially.

Induction: Assume that, $T(k) \leq c.2^k \;\; \forall \; k < n$

$\therefore T(n-1) \leq c.2^{n-1}$

and

$T(n-2) \leq c.2^{n-2}$ also true

Now,

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

$$\leq c.2^{n-1} + c.2^{n-2} + 1$$

$$= \frac{c2^n}{2} + \frac{c2^n}{4} + 1$$

$$= \frac{3}{4} c2^n + 1$$

$$= c.2^n - \frac{1}{4} c2^n + 1$$

$$= c.2^n - \left(\frac{1}{4} c2^n - 1\right)$$

$$\leq c.2^n \text{ proved}$$

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

CamScanner

⊛. Asymptotic Notations: [Imp]

6 Complexity analysis of an algorithm is done in terms of bound (upper bound & lower bound). For this purpose we need the concept of asymptotic notations.

i) Big Oh (O) notation: When we have only asymptotic upper bound then we use O notation. Mathematically, a function $f(x)$ is said to be Big Oh of another function $g(x)$ [i.e, $f(x) = O\,g(x)$], iff there exist two constants x_0 and c such that

$$f(x) \leq c * g(x) \quad \forall x \geq x_0.$$

When $f(x) = O\,g(x)$ then we say that $g(x)$ is the upper bound of $f(x)$.

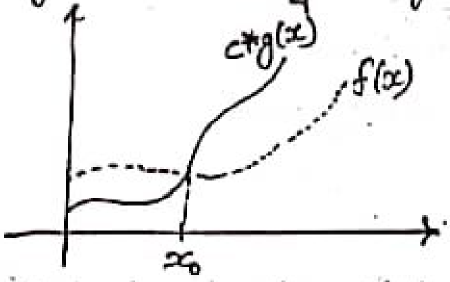


Fig: Geometrical interpretation of $f(x) = O\,g(x)$.

Example:- Find big oh of given function $f(n) = 3n^2 + 4n + 7$

Solution:- We have, $f(n) = 3n^2 + 4n + 7 \leq 3n^2 + 4n^2 + 7n^2 \leq 14n^2$

$$\Rightarrow f(n) \leq 14n^2$$

where, $c = 14$ and $g(n) = n^2$, thus $f(n) = O(g(n)) = O(n^2)$.

ii) Big Omega (Ω) notation:- Big omega notation gives asymptotic lower bound. If f and g are any two functions from set of integers to set of integers, the function $f(x)$ is said to be big omega of $g(x)$ i.e, $f(x) = \Omega(g(x))$ if and only if there exists two positive constants c and x_0 such that

For all $x >= x_0$, $f(x) >= c * g(x)$.

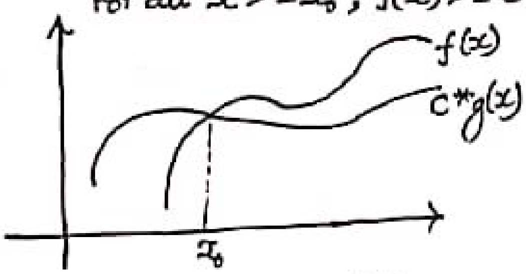


Fig: Geometric intrepretation of Big Omega notation.

When $f(x) = \Omega\,g(x)$ then we say that $g(x)$ is the lower bound of $f(x)$.

Example:- Find big omega of $f(n) = 3n^2 + 4n + 7$

Solution: Since we have $f(n) = 3n^2 + 4n + 7 >= 3n^2$

$$\Rightarrow f(n) >= 3n^2$$

where, $c = 3$ and $g(n) = n^2$, thus $f(n) = \Omega(g(n)) = \Omega(n^2)$.

⊛ Detailed Analysis of Algorithms:-

no need to remember each line just understand. Writing this is not asked in exam but concept is important

1) Time Complexity: (Analysis)

→ Time complexity of simple operations that takes 1 step time like assignment (e.g. $i=0$), addition (e.g. $a=b+c$), simple statements like
7 printf, scanf, return etc. take very small constant time, which does not affect time complexity of our algorithm much so we can neglect them.

Space or time complexity analyse करते वक्त term हटाएं ऐसे हटा complexity निकालते है worst case

→ We mainly analyze time complexity of algorithms based on the loops like for loop, while loop etc. We may have many condition in this case some of the simple cases are as follows:

i) If loop is like `for(i=0; i<=n; i++)` i.e, loop is simply running from 0 to n and incrementing simply by 1. In this case time complexity will be $O(n)$.

ii) For nested loops e.g. two loops running simply as in ⓘ which are nested. In this case time complexity will be product of time complexity of each loop.

for e.g.
```
for(i=0; i<=n; i++)        —— O(n)
{
   for(j=0; j<=n; j++)     —— O(n)
      printf("Hello");
}
```

$$\text{Time complexity} = O(n) \times O(n) = O(n^2).$$

iii) For loops like `for(i=0; i<=n; i*5)` i.e, incrementing by multiplication. In this case time complexity $= O(\log_{\text{constant multiplier}} n) = O \log_5 n$.

2) Space Complexity: (Analysis)

Space complexity is the total memory refrences used by the algorithm.

→ If total memory refrences used by the algorithm is constant like 1, 2, 3, 4, 5, etc. then the space complexity will be $O(1)$.

→ If Array is taking n memory refrences then the space complexity will be $O(n)$.

Examples:- Find detailed analysis of following factorial algorithm.

```
#include <stdio.h>
int main()
{  int i, n, fact=1;
   printf("Enter a number to calculate its factorial \n");
   scanf("%d", &n);
   for (i=1; i<=n; i++)
         fact = fact*i;
   printf("Factorial of %d = %d \n", n, fact);
   return 0;
}
```

__Time Complexity Analysis__

The declaration statement takes 1 step time

Printf statement takes 1 step time.

Scanf statement takes 1 step time.

__In for loop__

$i=1$ takes 1 step

$i<=n$ takes $(n+1)$ step

$i++$ takes n step.

$fact = fact*i$ takes n step

Printf statement takes 1 step.

return statement takes 1 step.

$\Rightarrow$ So Total time complexity $= 1+1+1+1+n+1+n+n+1+1$

$= 2n+7$

$= O(1) \times O(n) + O(1)$ — Since time complexity of constant is $O(1)$

$= O(n) + O(1)$ — Smaller terms are always neglected when there are higher ones like n, n^2 etc

$= O(n)$.

__Space Complexity Analysis__

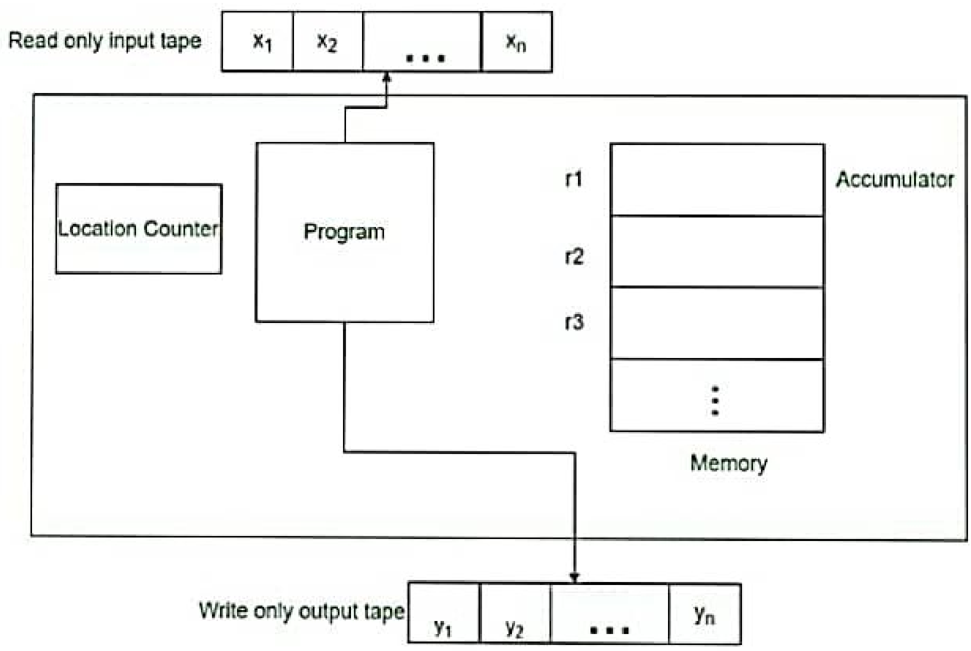
Total memory refrences used $= 3$ — i.e. Memory space is needed 1 for i, 1 for n and 1 for fact.

Hence, Space complexity $= O(1)$ — for constants $O(1)$

Random Access Machine (RAM)

8

Random Access Machine or RAM model is a CPU. It is a potentially unbound bank of memory cells, each of which can contain an arbitrary number or character. Memory cells are numbered and it takes time to access any cell in memory or say all operations (read/write from memory, standard arithmetic, and Boolean operations) take a unit of time. RAM is a standard theoretical model of computation (infinite memory and equal access cost). The Random Access Machine model is critical to the success of the computer industry.



RAM model: [Imp]

Random Access Machine (RAM) model is a model for counting the steps in algorithm in order to analyze the complexity. In this model we count:

- Basic operations (+, -, *, /) as 1 step.
- Memory refrence (read & write) as 1 step.
- Loops, function calls are not basic operations. Hence not counted.

$$Avg = \frac{a+b}{2}$$

Example: Algorithm to find the factorial of given numbers.

```
factorial (int n){
    if (n<0)
    fact = 1
    for(i=1; i<n; i++)
        fact = fact * i;
}return fact;
```

Step count according to RAM model:
$7n+3$

CamScanner

9 # Selection Sort

```
Selectionsort (A, n) {
        for (i=0; i<n-1 ; i++) {
                least : A[i];
                loc = i;
                for (j=i+1 ; j<n ; j++) {
                        if (n[j] <lead)
                        {
                            least = A[j];
                            loc = j;
                        }
                }
            swap(n[i], A[loc]);
}
```

Time Complexity:

when $i=0$, inner loop executes $(n=1)$ times
when $i=1$, inner loop executes $(n=2)$ times
when $i=2$, inner loop executes $(n=3)$ times
⋮
when $i=n-3$, inner loop executes 2 times
when $i=n-2$, inner loop executes 1 time

$\therefore T(n) = 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$

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

$$= \frac{n^2 - n}{2}$$

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

$$\leq \frac{1}{2} n^2$$

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

Bubble sort.

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

Time Complexity.

when $i=0$, inner loop executes $(n-1)$ times
when $i=1$, inner loop executes $(n-2)$ times
when $i=2$, inner loop executes $(n-3)$ times.

⋮

when $i=n-3$, inner loop executes 2 times.
when $i=n-2$, inner loops execute 1 times.

$\therefore T(n) = 1 + 2 + 3 \ldots + (n-3) (n-2) + (n-1)$

$= \frac{n(n+1)}{2}$ — formula for n natural n.

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

$= \frac{n^2}{2} - \frac{n}{2} = \frac{1}{2}n^2 \quad \therefore T(n) = O(n^2)$

classmate PAGE

DATE

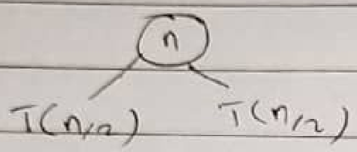
2 10 $T(n) = 2T(n/2) + n$ where $n > 1$ where $n = 1$.

* merge Sort
* Best case of Quick sort

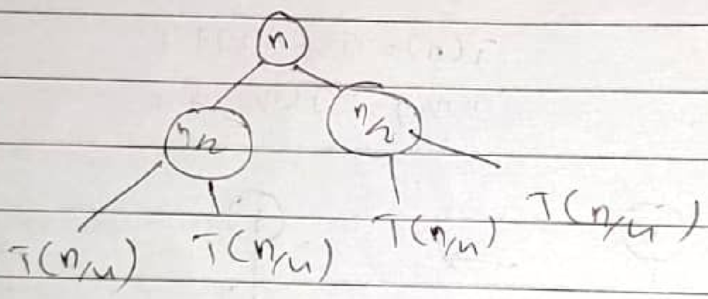
1st iteration

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

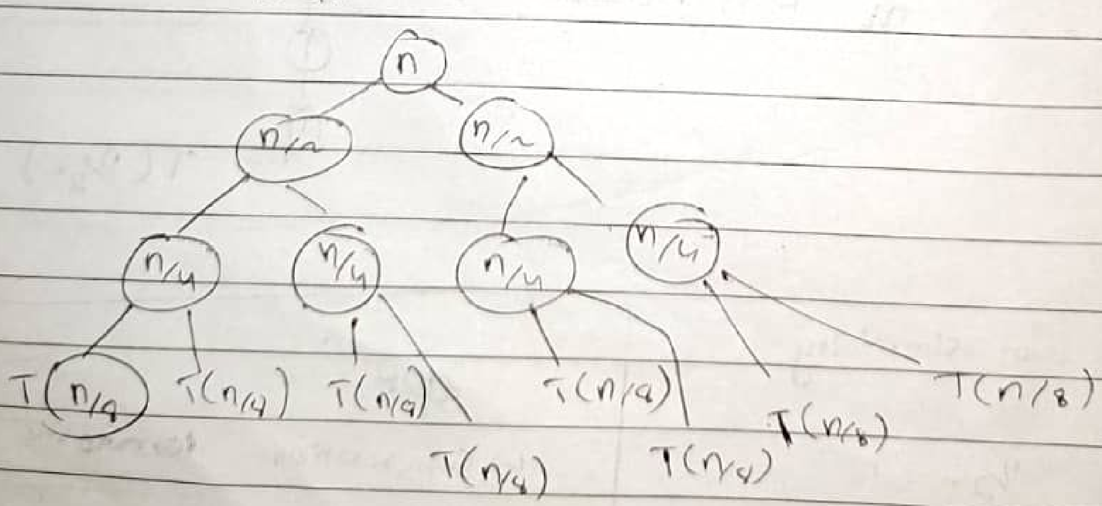
$T(n/4) = 2T(n/8) + n/4$



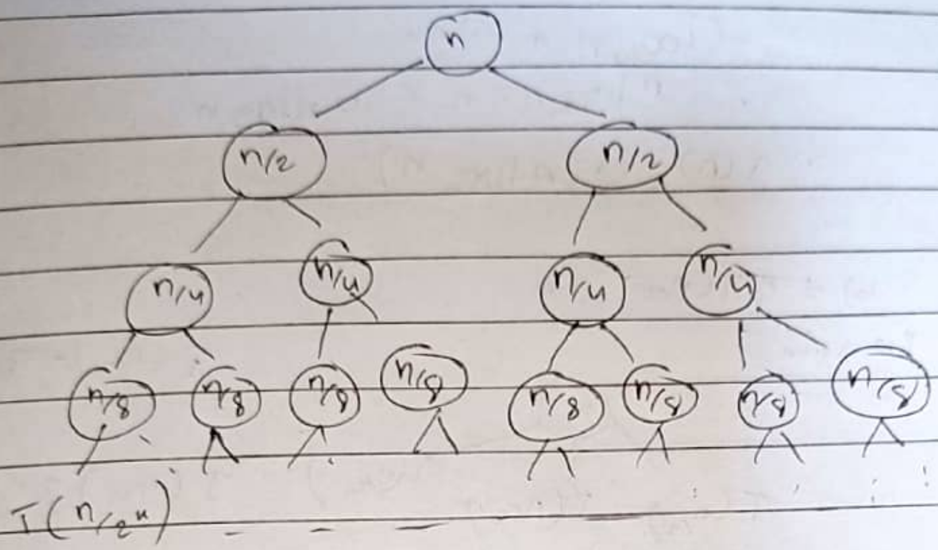
2nd iteration



3rd iteration



At k^{th} iteration

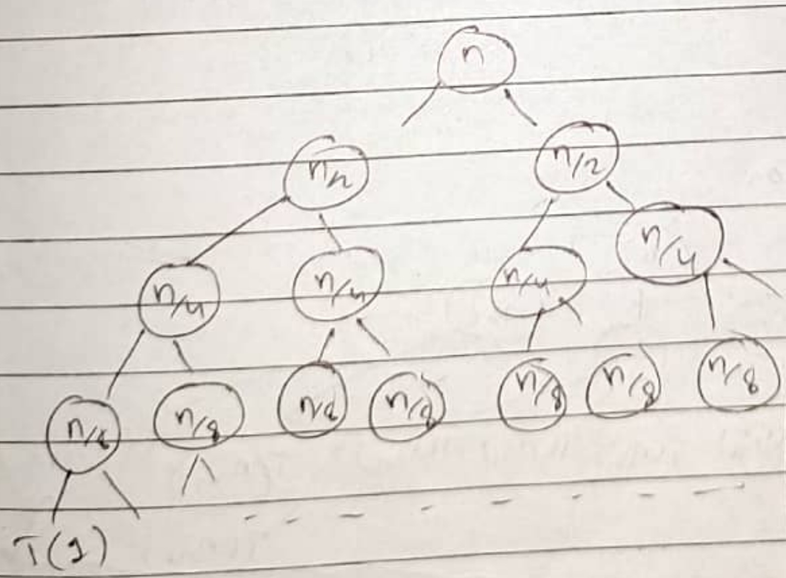


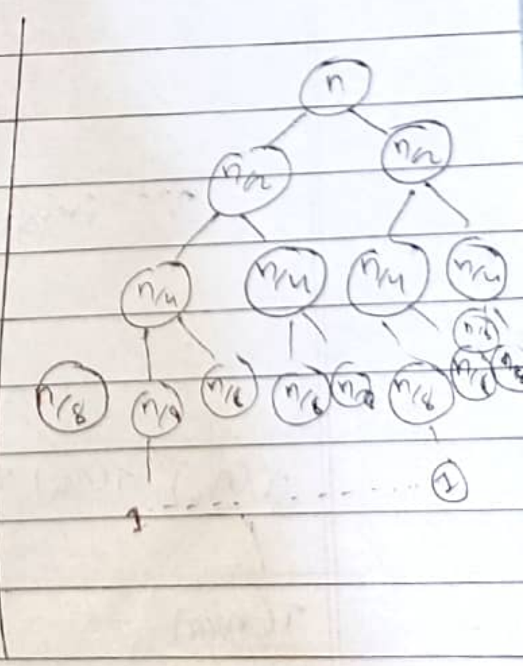
for simplicity.

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

$$n = 2^k$$

$$k = \log_2 n$$





$T(n) = n + n + n + n + \cdots + n$

$= (k+1)n$

$= (\log_2 n + 1)n$

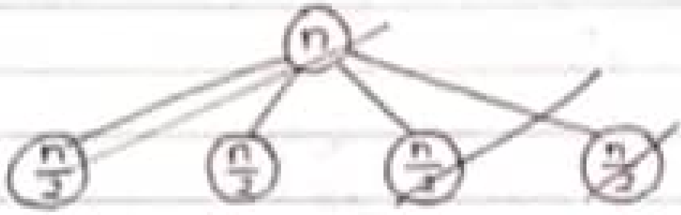
$= n\log_2 n + n \le 2n\log_2 n$

$\therefore T(n) = O(n\log_2 n)$

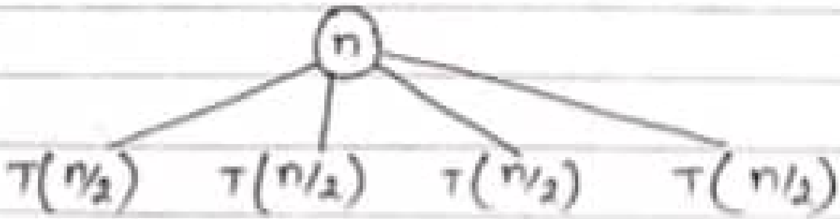
$T(n) = 4T(n/2) + n$

$\rightarrow$

1st iteration

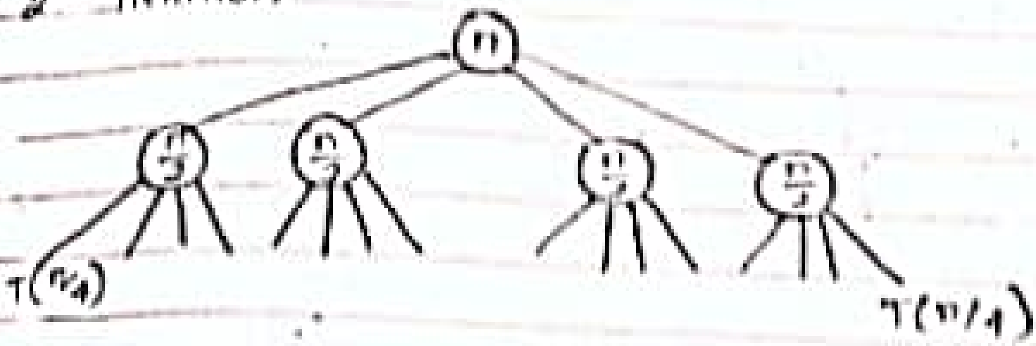


2nd iteration

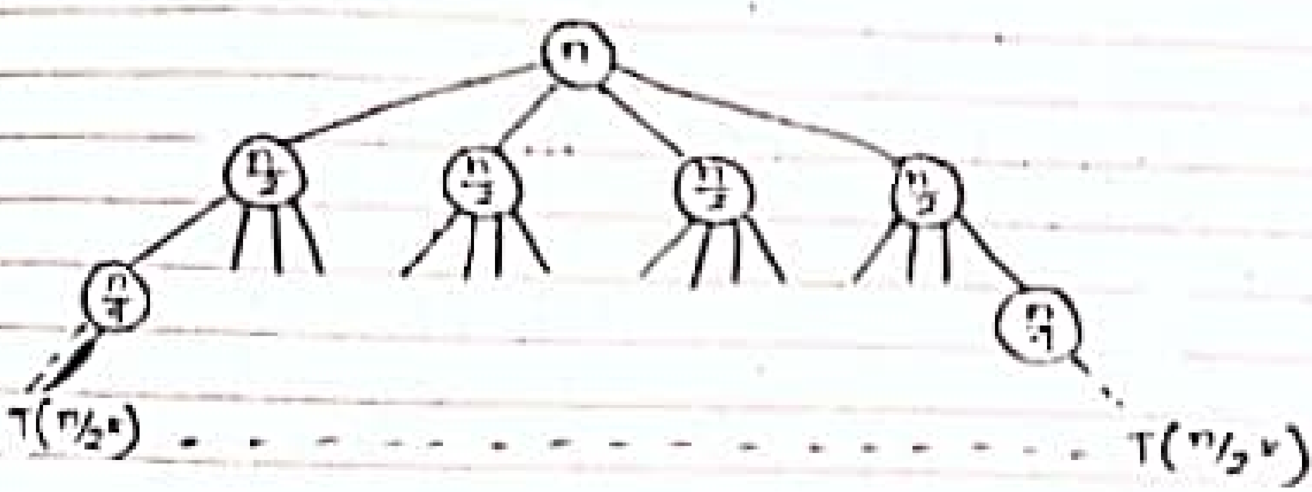


CamScanner

2nd iteration



At kth iteration

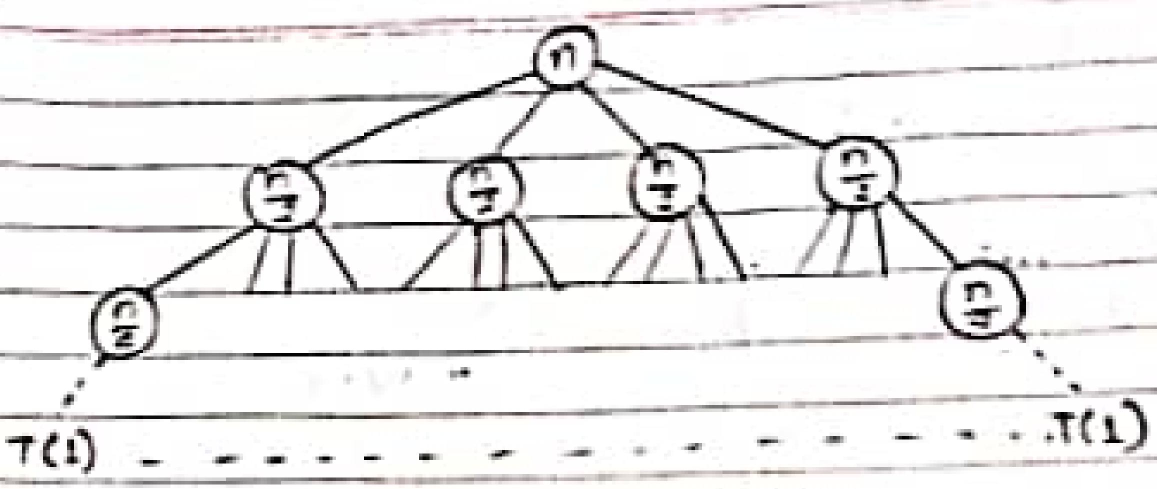


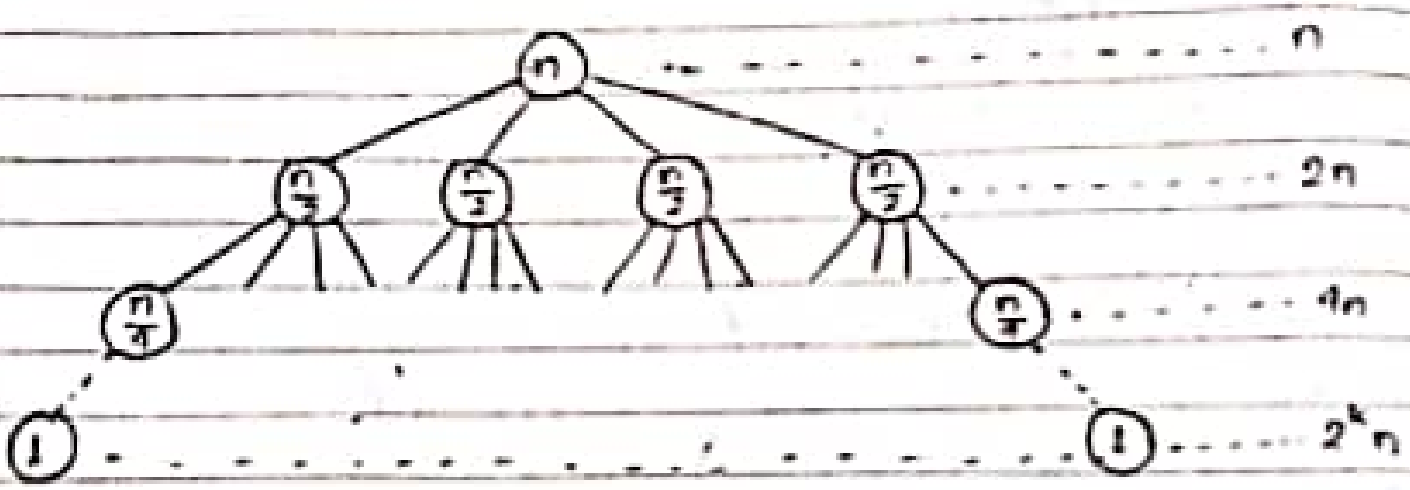
For simplicity

$n/2^k = 1$

$2^k = n$

$\boxed{k = \log_2 n}$





$T(n) = n + 2n + 4n + \dots + 2^k n$

$= n(1+2+4+\dots+2^k)$

$= n \cdot \dfrac{1(2^{k+1}-1)}{2-1}$

$= n\cdot(2\cdot 2^k - 1)$

$= n(2n-1)$

$= 2n^2 - n$

$\leq 2n^2$

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

$r = t_2/t_1$

$= 2/1$

$= 2 > 1$

$S_n = \dfrac{a(r^n-1)}{r-1}$ $r>1$

$r<1$

$S_n = \dfrac{a(1-r^n)}{1-r}$

9–10

⊛ Recurrence Relations:-

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a recurrence relation means to obtain a function defined on the natural numbers that satisfy the recurrence. To solve recursive algorithms we need to define their recurrence relation and by using any one of the recurrence relation solving method we calculate their complexity.

Example: Recursive algorithm for finding factorial.

$$T(n) = 1 \quad \text{when } n = 1$$

$$T(n) = T(n-1) + O(1) \quad \text{when } n > 1.$$

⊛ Solving Recurrences: [Imp],

Numericals are important. Recursion tree & Master method are more important

The process of finding solution of given recurrence relation in terms of big oh notation is called solving recurrences. There are a lot of methods for solving recurrence relations some of the popular methods are: Iteration method, Recursion Tree, Substitution and Master method.

1) Iteration method:

Here we expand the given relation until the boundry is not met. Expand the relation so that summation independent on n is obtained. It uses an initial guess to generate a sequence of improving approximate solutions for a class of problems, in which the nth approximation is derived from the previous ones.

2) Recursion Tree:

Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded. In general, we consider second term in recurrence as root. Each root and child represents cost of single sub-problem. Summing the cost at each level we determine the total cost.

3) Substitution method:-

The substitution method for solving recursences is described using two steps:

i) Guess the form of the solution.

ii) Use induction to show that the guess is valid.

Note: Initially guessing the solution of a problem depends on your practices.

4) Master Method:

Master Method is a direct way to get solution. The master method works only for recurrences that can be transformed to following type:

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

where, $a \geq 1, b > 1$ are constant, $f(n)$ asymptotically positive function. If the recurrence relation is in this form then there are following four possible cases occurred:

i) Master Method Case 1:

If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constants $\varepsilon > 0$ then,

$$T(n) = O(n^{\log_b a})$$

ii) Master Method Case 2:

If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constants $\varepsilon > 0$ then,

$$T(n) = O(f(n)).$$

iii) Master Method Case 3:

If $f(n) = \Theta(n^{\log_b a})$ for some constants $\varepsilon > 0$ then,

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

iv) Master Method Case 4:

In this case the master method cannot be applied.