Unit-1: DAA
Question

Section-A

1. What is the time complexity of a binary search algorithm?
2. What is an algorithm?
3. Write the recurrence relation that arises in relation with the complexity of binary search. Solve T(n)= 2 T(n/2) + n .
4. An algorithm is made up of 2 modules M1 and M2. If order of M1 is f(n) and M2 is g(n) then what is the order of the algorithm ?
5. What is the time complexity of an insertion sort algorithm in all cases?
6. Write Heap Sort algorithm.
7. What is Pseudo code?
8. Discuss any one sorting algorithm having linear time complexity.
9. What is a stable sort? Name any two stable sort algorithms.
10. Explain the term analysis of algorithm.
11. Write the Master Theorem.
12. What is heap and max heap?
13. Write an algorithm for counting sort.
14. Discuss the analysis of Heap sort.
15. Write an algorithm for Randomized Quick Sort.
16. Write the algorithm for Quick sort.
17. You are given an array of n integers a1< a2 <a3 ……<an. Give an O(log n) Algorithm that finds index i where ai=i or prove that i does not exist.
18. Show that 2n+1=O(2n).
19. Write an algorithm for Bucket Sort.

Section-B

1. Solve the following recurrences : T(n) = T(α.n) + T((1-α).n)+n, 0<α<1
2. Considering T(n) = 2T(n/2) + n2, we have to obtain the asymptotic bound using recursion tree method.
3. Find the solution of the following recurrence relation :
   a) T(n) = 8T(n/2) + 3n2 , where n>1.
   b) T(n) = T(n-1) + n, where n>1
4. Illustrate the functioning of Heap sort on the following array : A = {25, 57, 48, 37,12, 92, 86}
5. Illustrate the operation of Counting sort on the array A = {6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2}
6. Write an algorithm for counting sort.
7. Discuss the analysis of Insertion Sort.
8. Write an algorithm for Randomized Quick Sort.
9. Use a recursion tree to give an asymptotically tight solution to the recurrence T(n) = T(n-a) + T(a) + cn , where a>=1 and c>0
10. Write the Merge Sort Algorithm.

Section-C

1. Write the algorithm for Quick sort. Prove that the running time complexity of Quick sort is O(nlogn) in average.
2. Determine the asymptotic order of the following functions:
   (i) f(n)=3n2 + 5
   (ii) f(n)=2n + n +3
   (iii) f(n)=5
   (iv) f(n)=n + 7
3. Find the solution of the following recurrence relation using recursion tree method T(n) = T(n/3) + T(2n/3) + O(n)
4. Why do we use asymptotic notation in the study of algorithms? Explain in brief various asymptotic notations and give their significance.
5. Solve the following recurrences. .
   (i)T(n) = 2T(n1/2)+1
   (ii)T(n)=5T(n/5)+n/logn
6. Solve the following By Recursion Tree Method
   T(n) = T(n/5) + T(4n/5) + n

==========

ANSWERS

Section-A

1. Time Complexity of the Binary Search Algorithm is **$O(\log_2 n)$**.

2. *Algorithm is a set of steps to complete a task.*

   For example, Task: to make a cup of tea. Algorithm:
   a. add water and milk to the kettle,
   b. boilit, add tea leaves,
   c. Add sugar, and then serve it in a cup.

   An **algorithm** is any well-defined computational procedure that takes some value,or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

3. The recurrence relation that arises in relation with the complexity of binary search is:

$$T(n) = T\left(\frac{n}{2}\right) + k, \text{ k is a constant}$$

   (Solve it by master method and you will get **$\Theta(\log n)$**)
   Solve 2nd part of question by master theorem, it will come $\Theta(n \log n)$

4. The order of the algorithm is **max(f(n), g(n))**.

5. Time complexity of an insertion sort algorithm is **Θ(** n^2 **)** in **worst** and **average case** and **Θ(n)** in **best case**.

6. Heap Sort algorithm :-

```
Heapsort(A)
{
      Buildheap(A)
      for i ← length[A] down to 2
         do swap A[1] ←→ A[i]
         heapsize[A] ← heapsize[A] - 1
         Heapify(A, 1)
}
```

```
Buildheap(A)
{
    heapsize[A] ←length[A]
    for i ←⌊length[A]/2⌋ down to 1
       do Heapify(A, i)
}
```

```
Heapify(A, i)
{
     l ← left(i)
     r ← right(i)
     if l <= heapsize[A] and A[l] > A[i]
        then largest ←l
        else largest ← i
     if r <= heapsize[A] and A[r] > A[largest]
        then largest ← r
     if largest != i
        then swap A[i] ←→ A[largest]
           Heapify(A, largest)
}
```

7. Pseudo code :-
   - High-level description of an algorithm
   - More structured than English prose
   - Less detailed than a program
   - Preferred notation for describing algorithms
   - Hides program design issues

8. Counting Sort :-

Counting Sort

Input: $A[1..n]$,
$A[J] \in \{1,2,\ldots,k\}$

Output: $B[1..n]$,
sorted

Uses $C[1..k]$,
auxiliary storage

```
Counting-Sort( A, B, k)
 1. for i ← 0 to k
 2.    do     C[i ] ← 0
 3. for j ← 1 to length[A]
 4.    do     C[A[ j ] ] ← C[A[ j ] ] + 1
 5. for i ← 1 to k
 6.    do     C[i ] ← C[i ] +C[i -1]
 7. for  j ←  length[A] down 1
 8.    do     B [ C[A[ j ] ] ] ← A[ j ]
 9.           C[A[ j ] ] ] ← C [A[ j ] ] -1
```

Analysis:

- **Assumptions:**
  - n records
  - Each record contains keys and data
  - All keys are in the range of 1 to k
- **Space**
  - The unsorted list is stored in A, the sorted list will be stored in an additional array B
  - Uses an additional array C of size k
- Main idea:
  - For each key value i, i = 1,…,k, count the number of times the keys occur in the unsorted input array A. Store results in an auxiliary array, C
  - Use these counts to compute the offset. $Offset_i$ is used to calculate the location where the record with key value i will be stored in the sorted output list B. The $Offset_i$ value has the location where the last Key_i .
- Analysis:
  - O(k + n) time
  - This is a stable sort: It preserves the original order of equal keys.

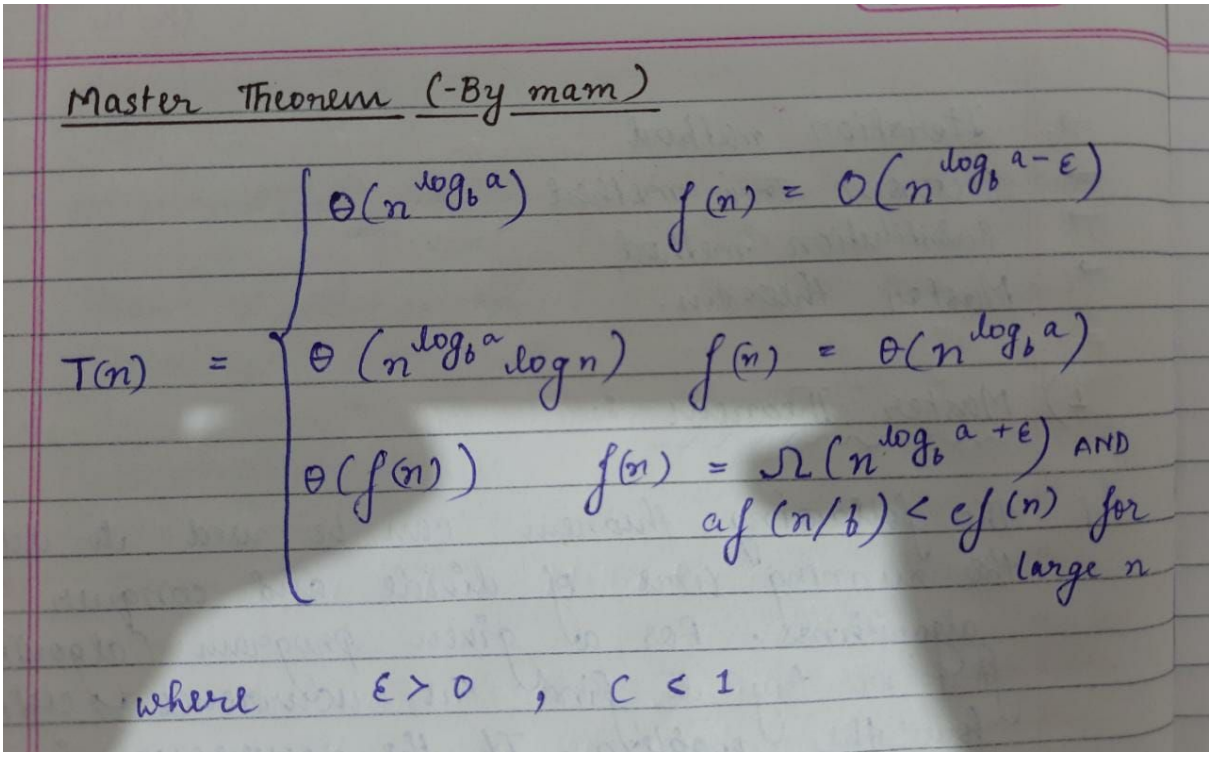
9. **Stable sorting** algorithms maintain the relative order of records with equal keys (i.e. values). That is, a **sorting** algorithm is **stable** if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the **sorted** list.
   (OR)
   A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

-> Some Sorting Algorithms are stable by nature like **Insertion Sort, Merge Sort** and **Bubble Sort** etc.

10. **Analyzing an algorithm** has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer.
    Running Time of an algorithm is the time taken by the algorithm to execute successfully. The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine independent as possible.

11. In divide and conquer algorithm. An algorithm that divides the problem of size n into a subproblems, each of size n/b

    Then, the Master Theorem gives us a method for the algorithm's running time:



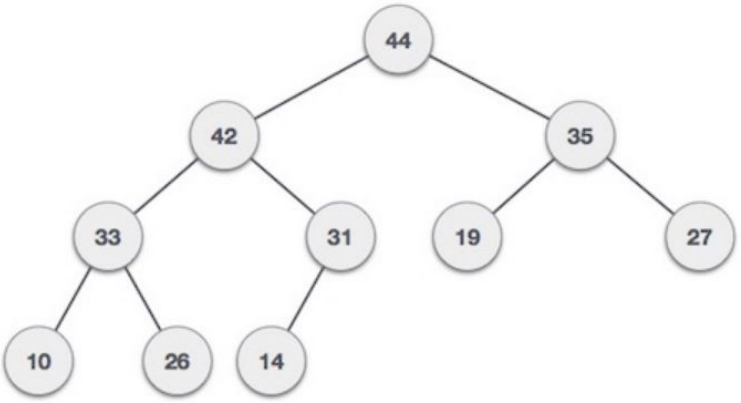
12. Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then

    key(α) ≥ key(β)

    As the value of a parent is greater than that of a child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types −

    For Input → 35 33 42 10 14 19 27 44 26 31

Max-Heap − Where the value of the root node is greater than or equal to either of its children.



13. There is no question 13 😒

14. Counting sort algorithm :-

Counting Sort

Input: $A[1..n]$,
$A[J] \in \{1,2,\ldots,k\}$

Output: $B[1..n]$,
sorted

Uses $C[1..k]$,
auxiliary storage

```
Counting-Sort( A, B, k)
 1. for i ← 0 to k
 2.    do     C[i ] ← 0
 3. for j ← 1 to length[A]
 4.    do     C[A[ j ] ] ← C[A[ j ] ] + 1
 5. for i ← 1 to k
 6.    do     C[i ] ← C[i ] +C[i -1]
 7. for  j ←  length[A] down 1
 8.    do     B [ C[A[ j ] ] ] ← A[ j ]
 9.           C[A[ j ] ] ] ← C [A[ j ] ] -1
```

15. Analysis of heap sort :-

- Build Heap Algorithm will run in O(n) time
- There are n-1 calls to Heapify each call requires O(log n) time
- Heap sort program combine Build Heap program and Heapify, therefore it has the running time of O(n log n) time
- Total time complexity: O(n log n)

16. Algorithm for **Randomized Quick Sort** :-

```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo     // place for swapping
    for j := lo to hi - 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i

partition_r(arr[], lo, hi)
    r = Random Number from lo to hi
    Swap arr[r] and arr[hi]
    return partition(arr, lo, hi)

quicksort(arr[], lo, hi)
    if lo < hi
        p = partition_r(arr, lo, hi)
        quicksort(arr, p-1, hi)
        quicksort(arr, p+1, hi)
```

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array).

17. Refer to long 1(a).

18. Only write the below photo part and the name of algo i.e; **binary search**.

Solution 1: consider the array B with

$$B[i] = A[i] - i.$$

Since $A[i] < A[i+1]$, we get

$$B[i] = A[i] - i \leq A[i+1] - 1 - i = B[i+1],$$

---

so the array B is non-decreasing and sorted.

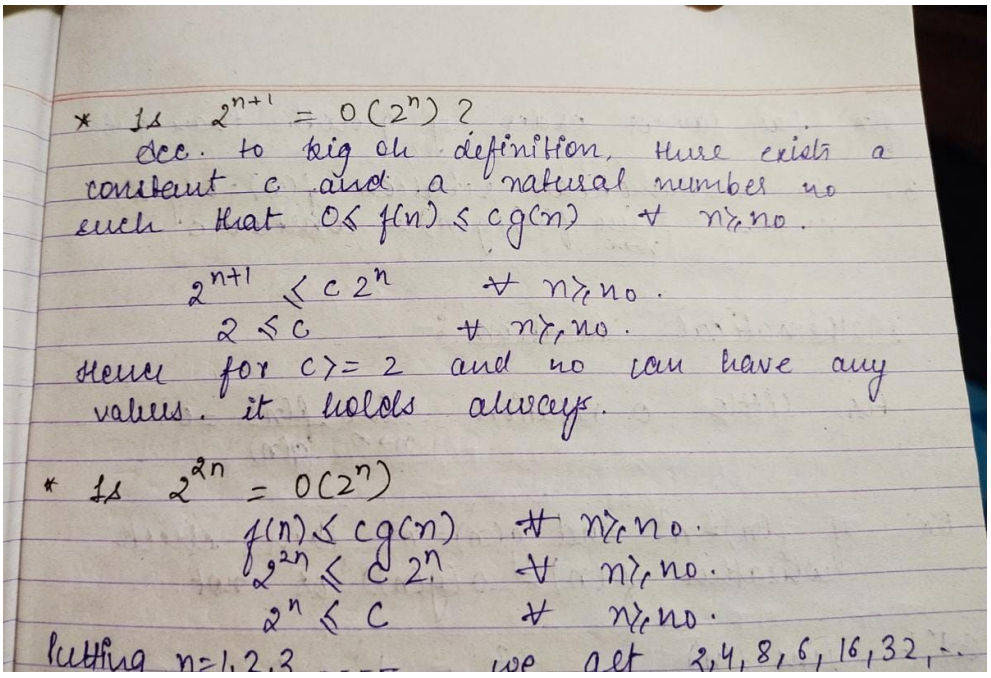
An index where $A[i] = i$ corresponds to one where $B[i] = 0$, so it suffices to binary search for 0 in B, which takes $O(\log n)$ time.

ALgorithm

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exist.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
```

end while
end procedure

19. Show that :-



20. **Bucket sort algo** :-

```
bucketSort(arr[], n)
1) Create n empty buckets (Or lists).
2) Do following for every array element arr[i].
.......a) Insert arr[i] into bucket[n*array[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.
```

Analysis :-

- P = 1/n , probability that the key goes to bucket i.
- Expected size of bucket is np = n 1/n = 1
- The expected time to sort one bucket is (1).
- Overall expected time is (n).

Bucket Sort algo (by mam) :- *(prefer this algo)*

- Keys are distributed uniformly in interval [0, 1]
- The records are distributed into n buckets
- The buckets are sorted using one of the well known sorts
- Finally the buckets are combined

==========

Section-B

1. **Solve the following recurrences : T(n) = T(α.n) + T((1-α).n)+n, 0<α<1**

We can assume that $0 < \alpha \leq 1/2$, since otherwise we can let $\beta = 1-\alpha$ and solve it for β.

Thus, the depth of the tree is $\log_{1/\alpha} n$ and each level costs cn. And let's guess that the leaves are $\Theta(n)$,

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_{1/\alpha} n} cn + \Theta(n) \\ &= cn \log_{1/\alpha} n + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

We can also show $T(n) = \Theta(n \lg n)$ by substitution.

To prove the upper bound, we guess that $T(n) \leq dn \lg n$ for a constant $d > 0$,

$$\begin{aligned} T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \\ &\leq d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + cn \\ &= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\ &= dn \lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \\ &\leq dn \lg n, \end{aligned}$$

where the last step holds when $d \geq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}$.

We can achieve this result by solving the inequality

$$\begin{aligned} dn \lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn &\leq dn \lg n \\ \implies dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn &\leq 0 \\ \implies d(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) &\leq -c \\ \implies d &\geq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}, \end{aligned}$$

To prove the lower bound, we guess that $T(n) \geq dn \lg n$ for a constant $d > 0$,

$$\begin{aligned} T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \\ &\geq d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + cn \\ &= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\ &= dn \lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \\ &\geq dn \lg n, \end{aligned}$$

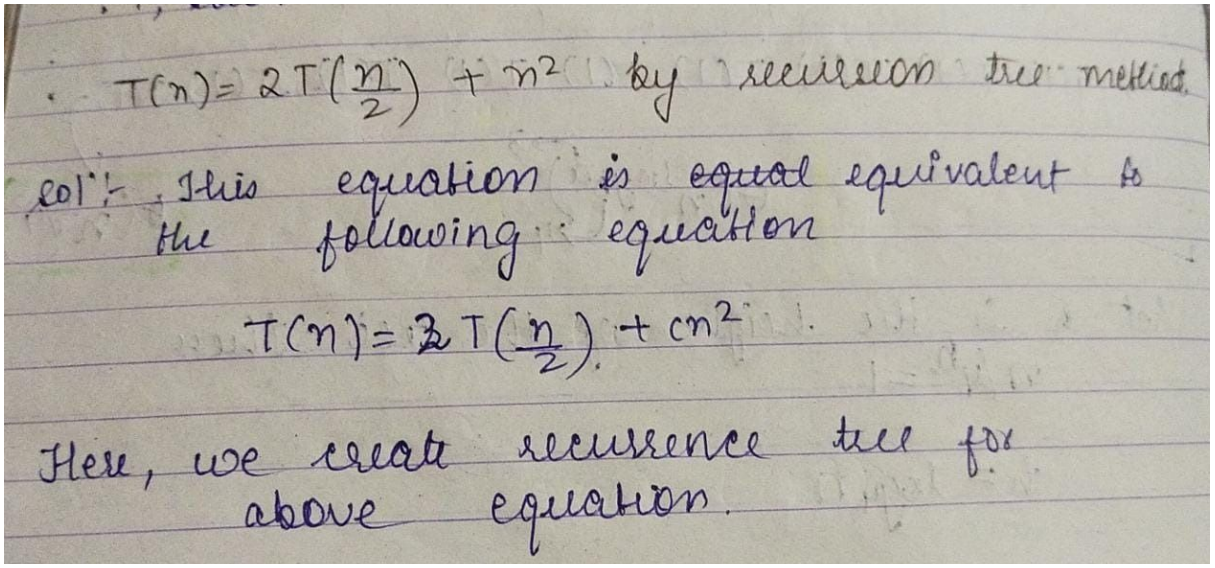
where the last step holds when $0 < d \leq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}$.

We can achieve this result by solving the inequality

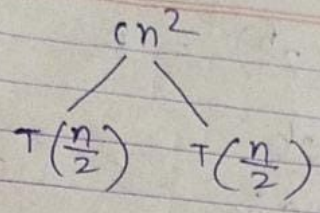
$$\begin{aligned} dn\lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn &\geq dn \lg n \\ \implies dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn &\geq 0 \\ \implies d(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) &\geq -c \\ \implies 0 < d &\leq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}, \end{aligned}$$

Therefore, $T(n) = \Theta(n \lg n)$.

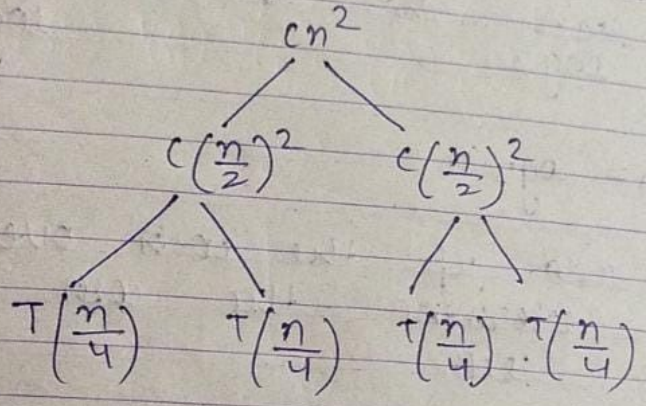
2. Considering T(n) = 2T(n/2) + n2 , we have to obtain the asymptotic bound using the recursion tree method.



$T(n)$

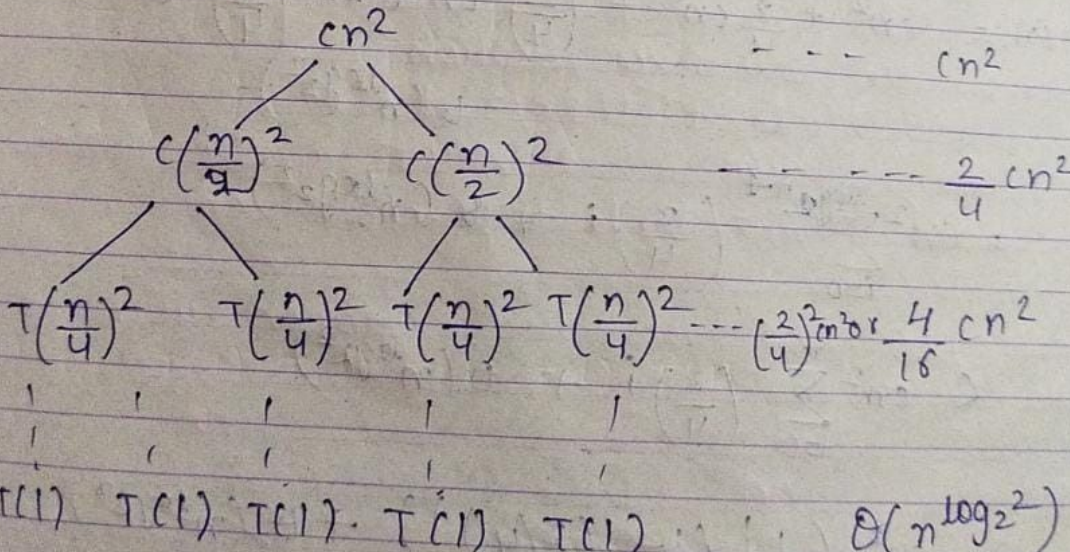


(a)



(b)

(c)



Let h is the height of tree, then.

$n/2^h = 1$.

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

$$\log(n) - \log(2^h) = 0$$

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

$$\log n = h \log 2$$

$$\frac{\log n}{\log 2} = h \quad \text{or} \quad h = \frac{\log n}{\log 2}$$

$$h = \log_2 n$$

Now we add up the costs over all levels to determine the costs for the entire tree:

$$T(n) = cn^2 + \frac{2}{4}cn^2 + \left(\frac{2}{4}\right)^2 cn^2 + \cdots \left(\frac{2}{4}\right)^{\log_2 n - 1} + \Theta(n^{\log_2 2})$$

$$= cn^2 \sum_{i=0}^{\log_2 n - 1} \left(\frac{2}{4}\right) i + \Theta(n^{\log_2 2})$$

$$< cn^2 \sum_{i=0}^{\infty} \left(\frac{2}{4}\right) i + \Theta(n^{\log_2 2})$$

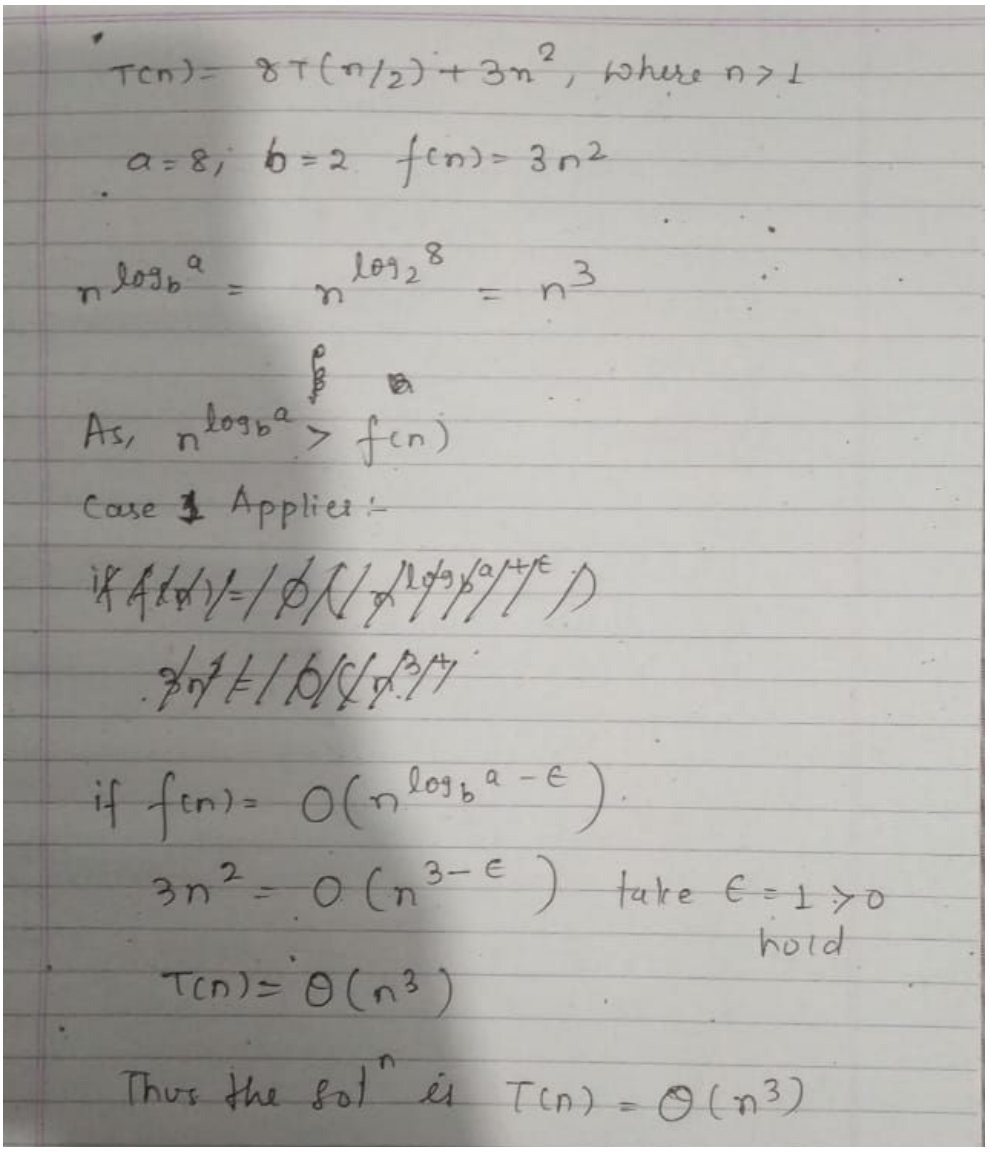
$$= cn^2 \left(1 / (1 - (2/4))\right) + \Theta(n^{\log_2 2})$$

$$= cn^2(2) + \Theta(n^{\log_2 2})$$

$$= O(n^2)$$

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

3. Find the solution of the following recurrence relation : a) T(n) = 8T(n/2) + 3n 2 , where n>1. b) T(n) = T(n-1) + n, where n>1



$T(n) = 8T(n/2) + 3n^2$, where $n > 1$

$a = 8;\ b = 2\ \ f(n) = 3n^2$

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

As, $n^{\log_b a} > f(n)$

Case 1 Applies :-

~~if $f(n) = O(n^{\log_b a + \epsilon})$~~

~~$3n^2 = O(n^{3+\epsilon})$~~

if $f(n) = O(n^{\log_b a - \epsilon})$

$3n^2 = O(n^{3-\epsilon})$ take $\epsilon = 1 > 0$ hold

$T(n) = \Theta(n^3)$

Thus the sol^n is $T(n) = \Theta(n^3)$

$$T(n) = T(n-1) + n \quad \text{—— (1)}$$

$\downarrow$

$n = n-1$

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

$$T(n-1) = T(n-2) + n-1 \quad \text{—— (2)}$$

$$T(n) = T(n-2) + n-1 + n \quad \text{—— (3)}$$

$n = n-2$ in Eq^{n} (1)

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

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

$$T(n) = T(n-3) + n-2 + n-1 + n \quad \text{—— (4)}$$

$n = k$

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

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

$$= T(0) + 1 + 2 + n$$

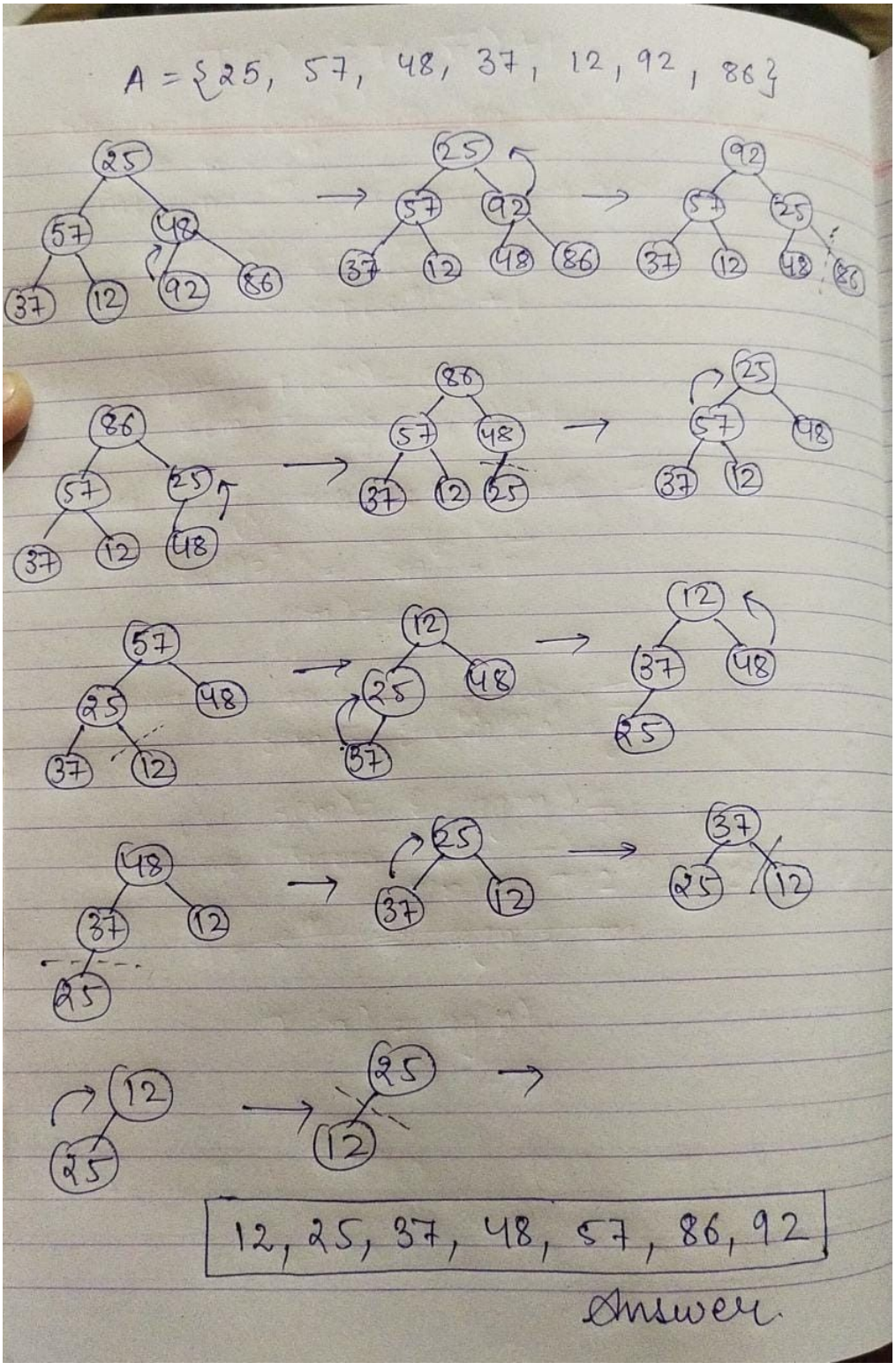
$$= 0 + 1+2+n$$

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

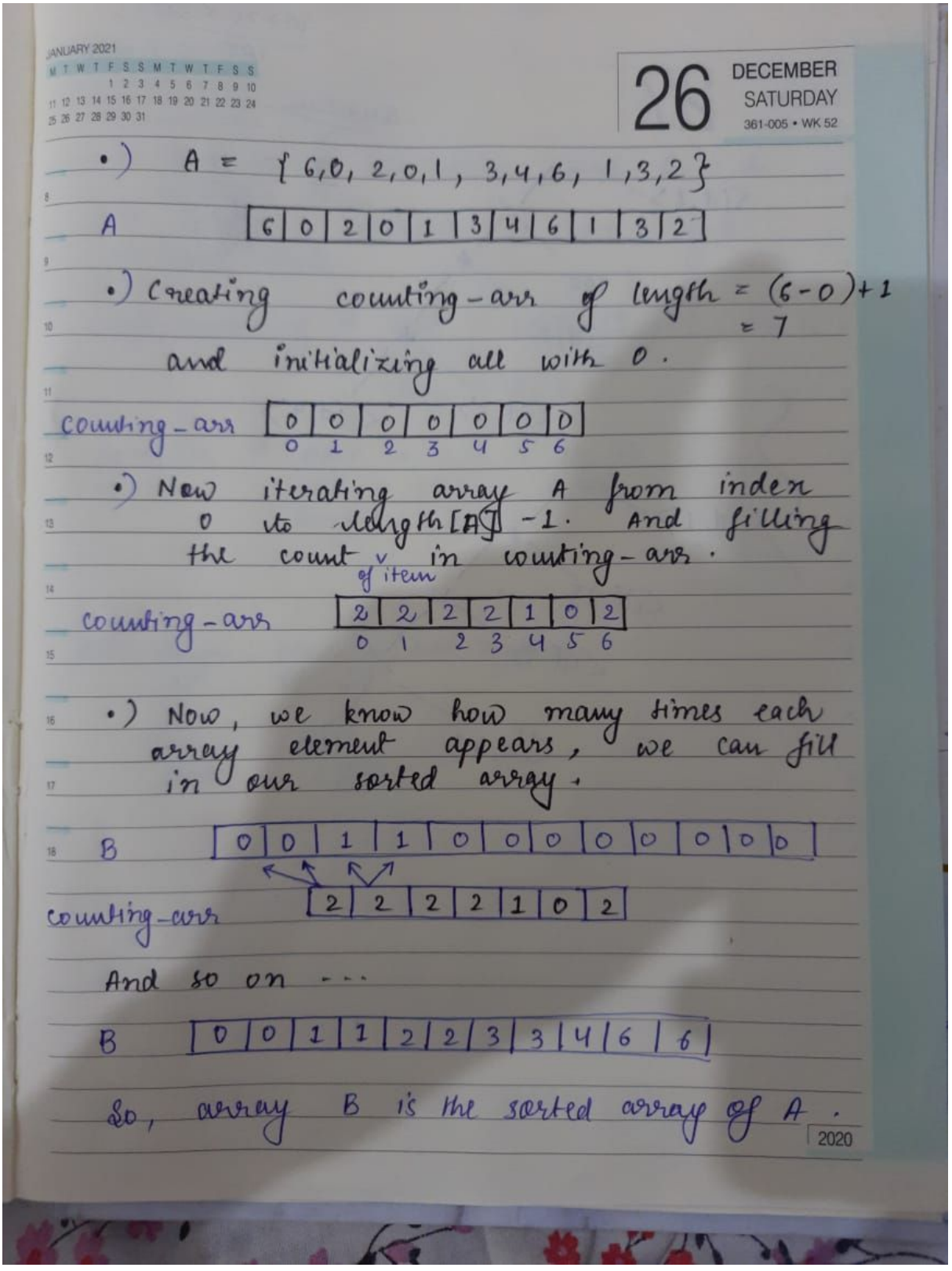
$$= n^2$$

Complexity $= O(n^2)$

4. Illustrate the functioning of Heap sort on the following array : A = {25, 57, 48, 37,12, 92, 86}



5. Illustrate the operation of Counting sort on the array A = {6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2}



6. Write an algorithm for counting sort.

Ans: refer to short **ques-8**

7. Discuss the analysis of Insertion Sort.

Analysis of Insertion Sort

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Best Case Analysis

- The array is already sorted "**while** i > 0 and A[i] > key"
  - A[i] ≤ key upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$ and $\sum_{j=2}^{n} t_j$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

  $= an + b = \Theta(n)$

Worst Case Analysis

- The array is in reverse sorted order — "**while** i > 0 and A[i] > key"
  - Always **A[i] > key** in **while** loop test
  - Have to compare **key** with all elements to the left of the **j**-th position hence t_j = j

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$= an^2 + bn + c$ a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

8. Write an algorithm for Randomized Quick Sort.

Ans: refer to short **ques-16**.

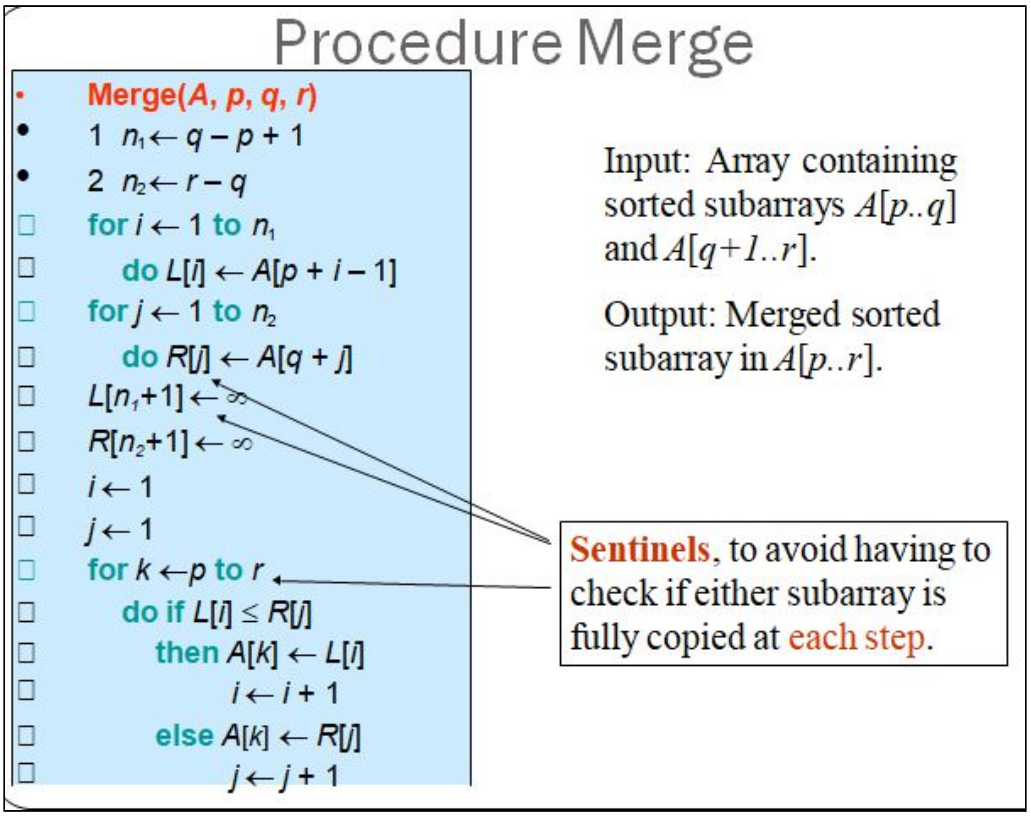
10. Write the Merge Sort Algorithm.

Merge-Sort (A, p, r)

- **INPUT: a sequence of *n* numbers stored in array A**

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
1  if p < r
2     then q ← ⌊(p+r)/2⌋
3          MergeSort (A, p, q)
4          MergeSort (A, q+1, r)
5          Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(*A*, 1, *n*)



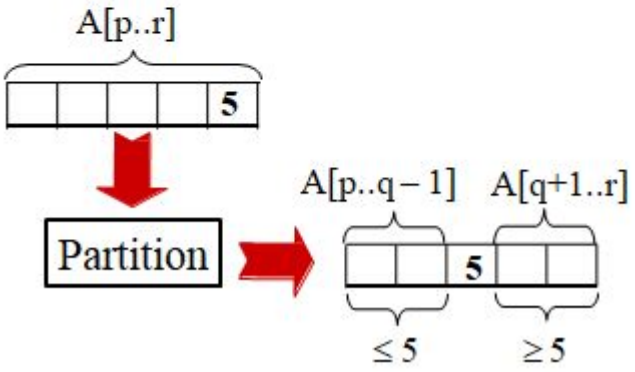
==========

Section-C

1. Write the algorithm for Quick sort. Prove that the running time complexity of Quick sort is O(nlogn) on average.

```
Quicksort(A, p, r)
    if p < r then
        q <- Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
    fi
```

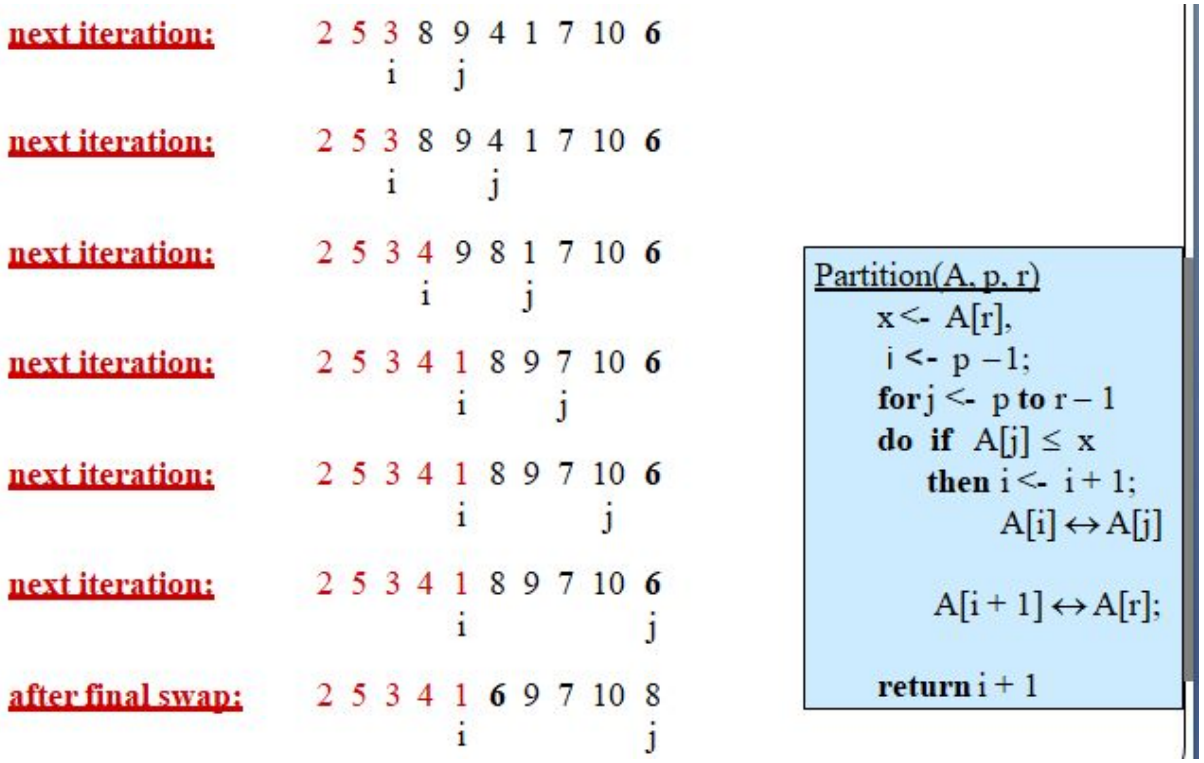
```
Partition(A, p, r)
    x <- A[r],
    i <- p – 1;
    for j <- p to r – 1
    do if A[j] ≤ x
        then i <- i + 1;
             A[i] ↔ A[j]

        A[i + 1] ↔ A[r];

    return i + 1
```



Example

initially:	p r 2 5 8 3 9 4 1 7 10 **6** i j	**note:** pivot (x) = 6
next iteration:	2 5 8 3 9 4 1 7 10 **6** i j	
next iteration:	2 5 8 3 9 4 1 7 10 **6** i j	
next iteration:	2 5 8 3 9 4 1 7 10 **6** i j	
next iteration:	2 5 3 8 9 4 1 7 10 **6** i j	

```
Partition(A, p, r)
    x <- A[r],
    i <- p – 1;
    for j <- p to r – 1
    do if A[j] ≤ x
        then i <- i + 1;
             A[i] ↔ A[j]

        A[i + 1] ↔ A[r];

    return i + 1
```



- Size of each subproblem $\leq n/2$.
  - One of the subproblems is of size $\lfloor n/2 \rfloor$
  - The other is of size $\lceil n/2 \rceil - 1$.
- Recurrence for running time
  - $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$

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

    By master method we get a=2 , b=2 , f(n)= n

- **$T(n) = \Theta(n \lg n)$**

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

By master method we get a=2 , b=2 , f(n)= n

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

$Since, f(n)=n$

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

Case 2 applies:

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

$$n = \Theta(n) \text{ satisfied}$$

$$\text{Thus the solution is } T(n) = \Theta(n \log n).$$

2. Determine the asymptotic order of the following functions:
   (i) f(n)=3n2 + 5
   (ii) f(n)=2n + n +3
   (iii) f(n)=5
   (iv) f(n)=n + 7

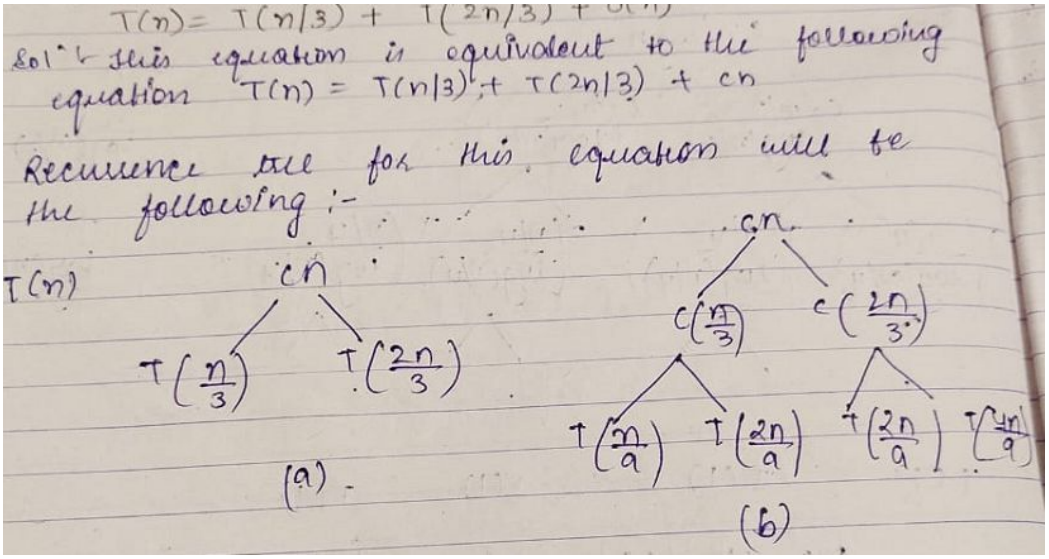
   ANS: (i) O(n^2)
   (ii)O(2^n)
   (iii)O(1)
   (iv)O(n)

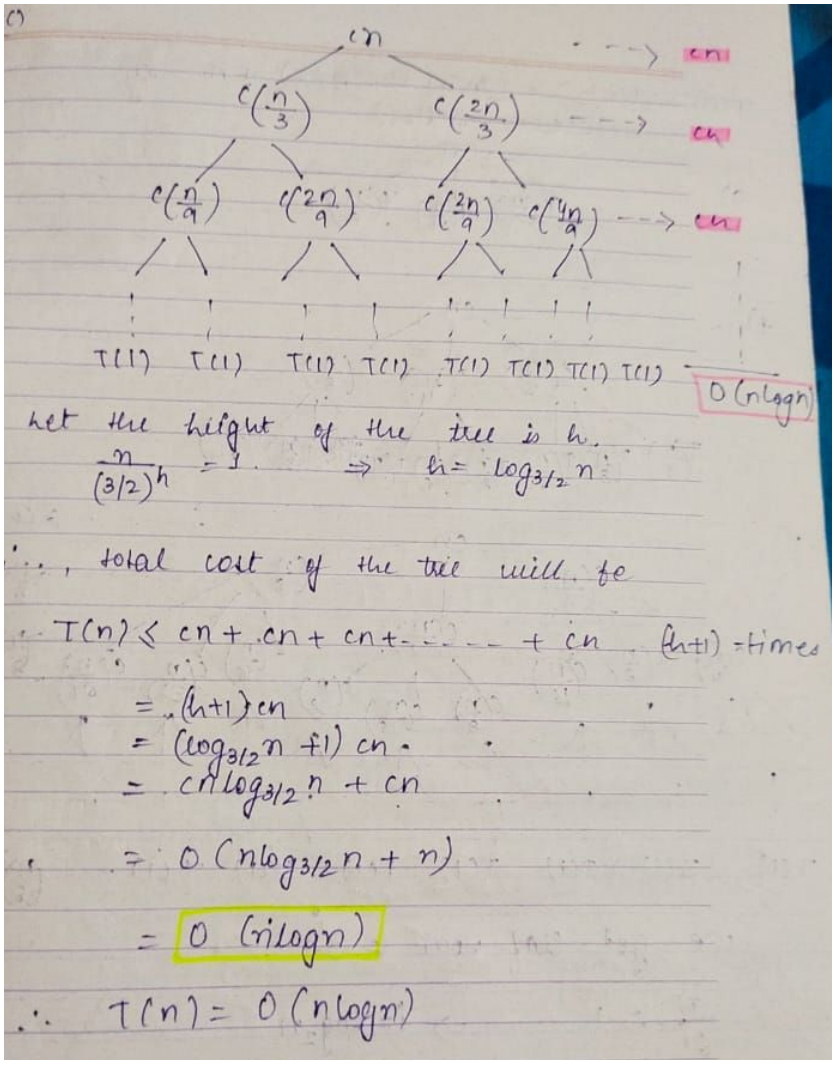
3. Find the solution of the following recurrence relation using recursion tree method T(n) = T(n/3) + T(2n/3) + O(n)

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

Sol: This equation is equivalent to the following equation $T(n) = T(n/3) + T(2n/3) + cn$

Recurrence tree for this equation will be the following :-





let the height of the tree is h.

$\frac{n}{(3/2)^h} = 1 \Rightarrow h = \log_{3/2} n$

∴, total cost of the tree will be

$T(n) \leq cn + cn + cn + \cdots + cn$ (h+1) times

$= (h+1)cn$

$= (\log_{3/2} n + 1)cn$

$= cn\log_{3/2} n + cn$

$= O(n\log_{3/2} n + n)$

$= O(n\log n)$

∴ $T(n) = O(n\log n)$

Write O(n $log_{3/2}$ n) and not just O(n log n)

4. Why do we use asymptotic notation in the study of algorithms?Explain in brief various asymptotic notations and give their significance.

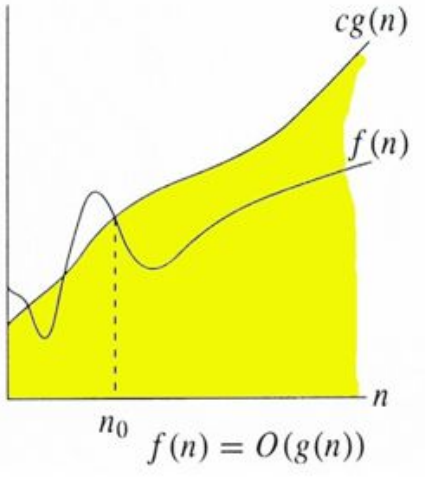
Asymptotic Notations are languages that allow us to analyze an **algorithm's** running time by identifying its behavior as the input size for the **algorithm** increases. This is also known as an **algorithm's** growth rate. **Asymptotic Notation** gives us the ability to answer these questions.

- When we study algorithms, we are interested in characterizing them according to their efficiency.
- We are usually interesting in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the *asymptotic running time*.
- We need to develop a way to talk about rate of growth of functions so that we can compare algorithms.
- *Asymptotic notation* gives us a method for classifying functions according to their rate of growth.

BIG-OH NOTATION (O)

Gives the upper bound of algorithm's running time. Let g(n) be a function.

O(g(n)) = {f(n) | there exists a constant c and a natural number n_0 such that that $0 \leq f(n) \leq c\ g(n)$ for all $n >= n_0$ }



SIGNIFICANCE :-

2. O-Notation (Upper bound) :

a. Big-oh is formal method of expressing the upper bound of an algorithm's running time.

b. It is the measure of the longest amount of time it could possibly take for the algorithm to complete.

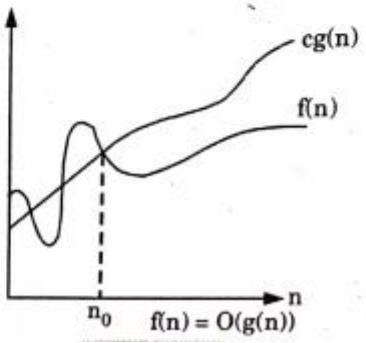
c. More formally, for non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$.

$$f(n) \leq cg(n)$$

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

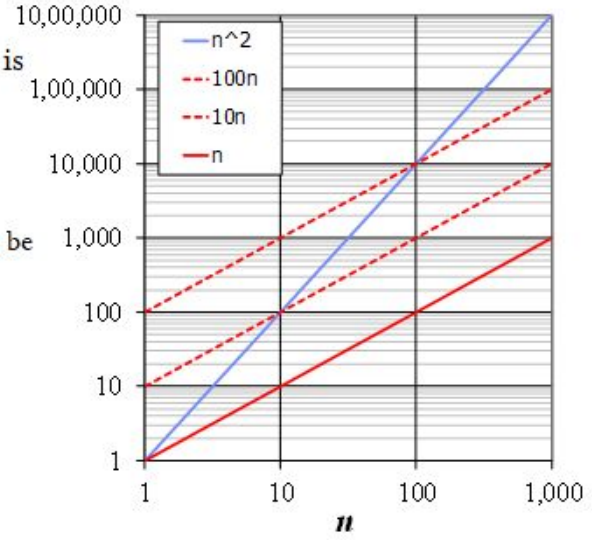
$$f(n) \in O(g(n))$$

i.e., the set of functions which, as n gets large, grow faster than a constant time $f(n)$.



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant

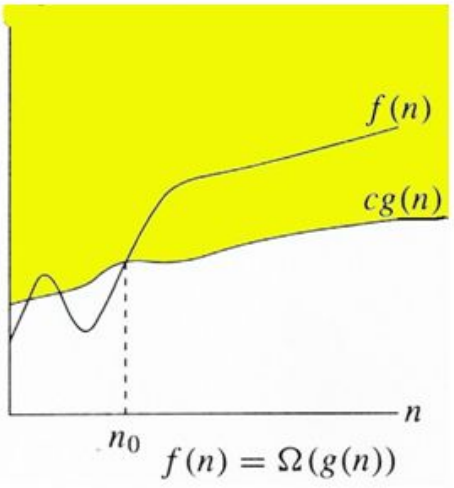


BIG-OMEGA NOTATION (Ω)

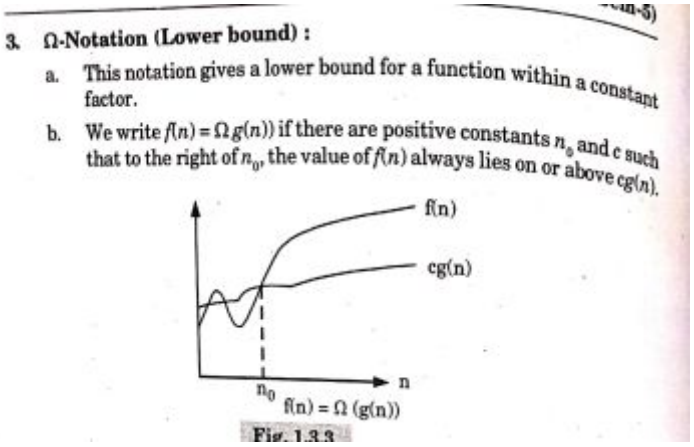
Gives the lower bound of algorithm's running time.

For a given function $g(n)$, we denote by $\Omega(g(n))$

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and $n0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0\}$.



SIGNIFICANCE :-



3. **Ω-Notation (Lower bound) :**

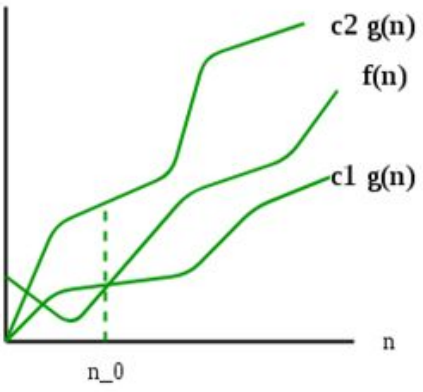
a. This notation gives a lower bound for a function within a constant factor.

b. We write $f(n) = \Omega g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above cg(n).

Fig. 1.3.3

THETA NOTATION (Θ)

This **notation** describes both upper bound and lower bound of an algorithm so we can say that it defines tight bound .For a given function g(n), we denote by Θ(g(n)) the set of functions and read as Θ of g(n).



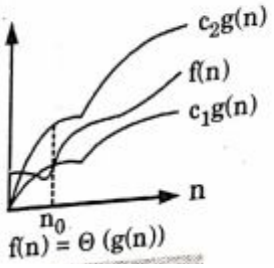
f(n) = theta(g(n))

It is defined as
Θ(g(n)) = {f(n) : there exist positive constants c1, c2, and n0 such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$}.

SIGNIFICANCE :-

Notations used

1. **Θ-Notation (Same order) :**
   a. This notation bounds a function within constant factors.
   b. We say $f(n) = \Theta g(n)$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.



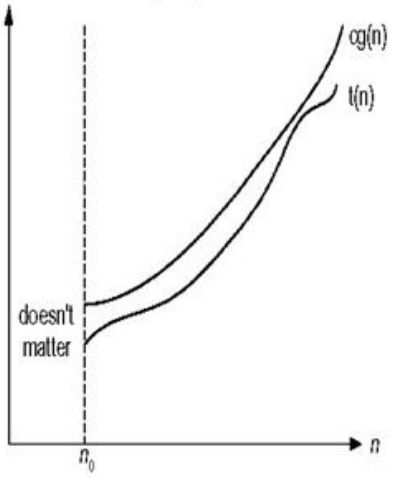
f(n) = Θ (g(n))

LITTLE OH NOTATION (o)

little-Oh Defn:

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

If for all positive constants c there exists an n_0 such that $f(n) < c \cdot g(n)$ for all $n >= n_0$.



SIGNIFICANCE :-

4. **Little-oh notation (o) :** It is used to denote an upper bound that is asymptotically tight because upper bound provided by O-notation is not tight.

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

5. **Little omega notation (ω) :** It is used to denote lower bound that is asymptotically tight.

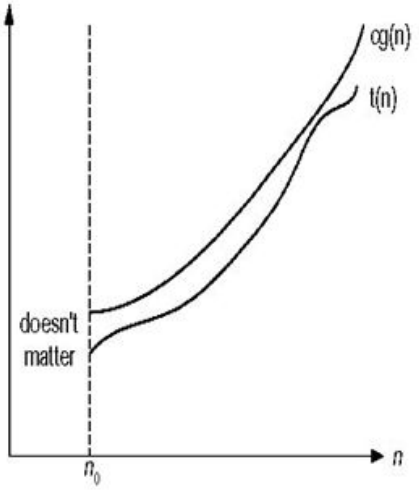
   $\omega(g(n)) = \{f(n)$: For any positive constant $c > 0$, if a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)\ \forall\ n \ge n_0\}$

LITTLE OMEGA NOTATION (ω)

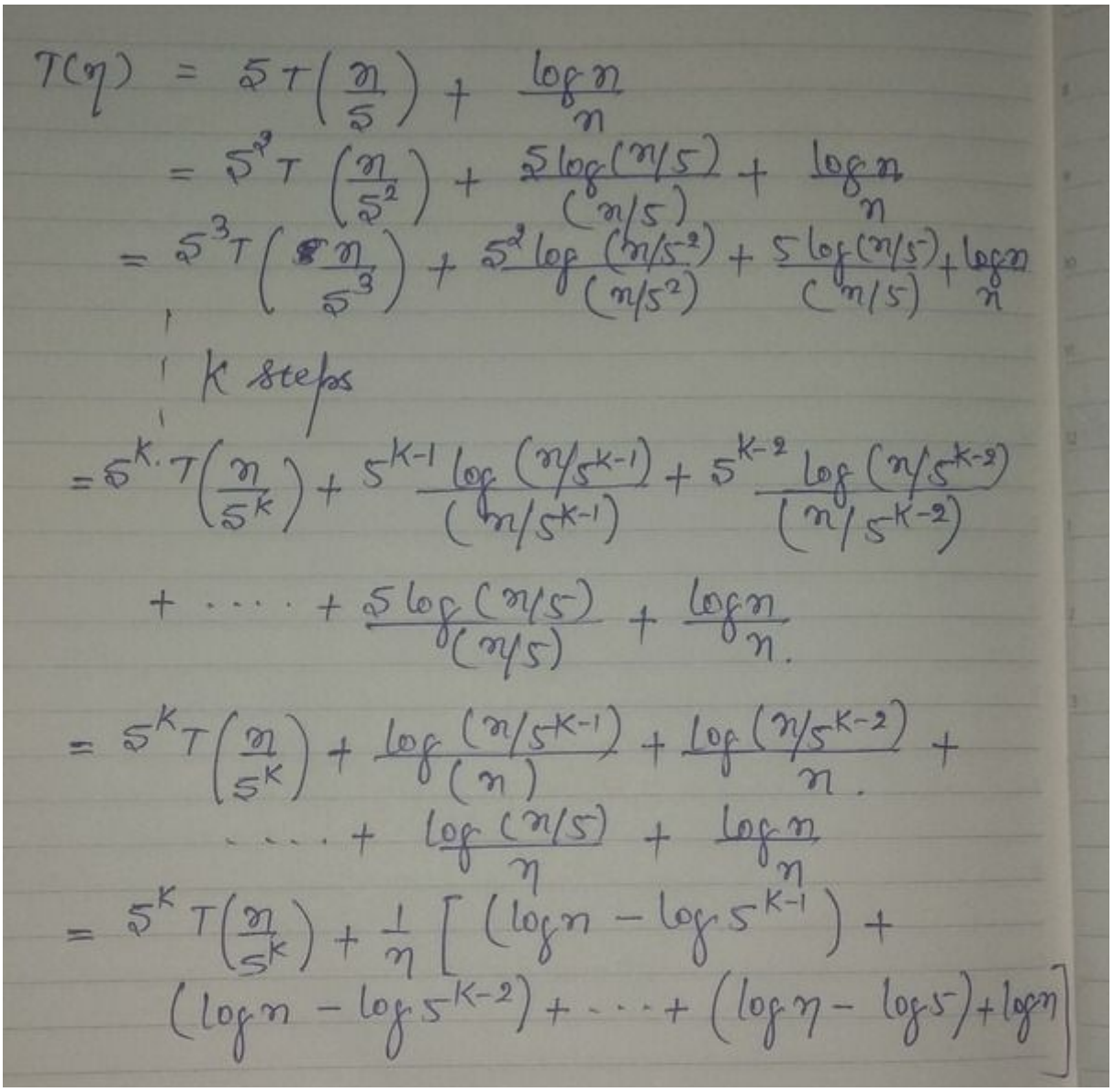
little-Omega Defn:

$f(n) = w(g(n))$

If for all positive constants c there exists an n_0 such that $c \cdot g(n) < f(n)$ for all $n \square n_0$.



5. Solve the following recurrences.
(i)T(n) = 2T(n1/2)+1
(ii)T(n)=5T(n/5)+n/logn



$$T(n) = 2T(\sqrt{n}) + 1$$

We will solve this by substitution method.

Let $n = 2^m$

Put $n = 2^m$ in given equation, we get

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

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

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

Since, by master method :-

$$m^{\log_2 2} = m^0 = 1$$
$$f(m) = 1.$$

Since, case 2 applies :-

$$S(m) = \theta(m^{\log_b a} \log m)$$

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

and we know that :

$$T(n) = T(2^m) = S(m) = \log(n)$$

~~$n = 2^m \Rightarrow \log n = m$~~

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

$$= 5^k T\left(\frac{n}{5^k}\right) + \frac{1}{n}\left[K \log n - \log 5^k + \log 5 . - \log 5^k + \log 5^2 + \cdots - \log 5^{-2} - \log 5\right]$$

$$= 5^k T\left(\frac{n}{5^k}\right) + \frac{1}{n}\left[K \log n - K \log 5^k\right]$$

Suppose, base condition is $T(1) = 1$

So, $\frac{n}{5^k} = 1$

$\Rightarrow n = 5^k$

$$= n + \frac{1}{n}\left[K (\log n)(\log n) - (\log n)(\log n)\right]$$

$$= n$$

6.

DECEMBER
FRIDAY
WK 52 • 360-006
25

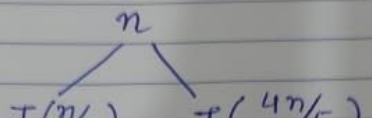
NOVEMBER 2020
M T W T F S S M T W T F S S
1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30

$T(n) = T(n/5) + T(4n/5) + n$

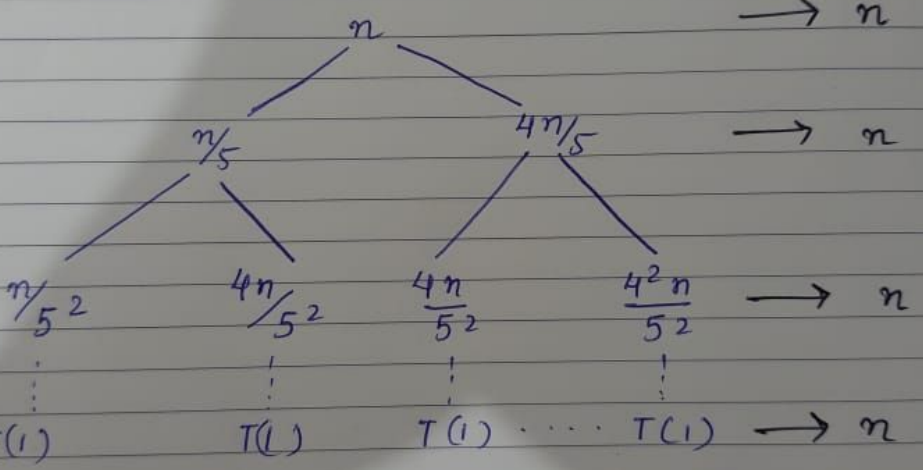
Now, we create recurrence tree for above eq^{n}s-

$T(n)$

(a)



(b)



Let height be $h \Rightarrow$ ~~h~~ $= 4n/5^{h} = 1$

$\Rightarrow h = \log_{5/4} n$

Total cost $\Rightarrow T(n) = n + (n + n + \cdots + n)$

$= n + n * h$

$= n + n \log_{5/4} n$

2020

$\therefore T(n) = \Theta(n \log_{5/4} n)$