Sorting Algorithms ① Comparision based Sorting algo ② Non Comparion based Sorting algo.

① Comparision based Sorting Algo:-

Sorting Algo	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity	Stable Sorting?	Inplace Sorting?	Algo Logic
① Quick Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n^2)$	No	Yes	Choose piviot and place at Correct position and Divide list base on piviot position.
② Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	Yes	No	Divide into two equal parts sort recursively and Merge into single Sorted list.
③ Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	No	Yes	Build Max heap && Delete Max $(n-1)$ times && Place max at end.
④ Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	Yes	Yes	Compare && Exchange Adjacency elements Repeat $(n-1)$ passes.
⑤ Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	No	Yes	Choose position of min from $a[K]$ to $a[n]$ and Swap with $a[K]$ where $K: 1,2,3,\cdots n-1$
⑥ Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	Yes	Yes	Place $a[i]$ into Correct position of Sorted part of array $a[1]$ to $a[i-1]$ where $i=2,3,\cdots n$

$\Omega(n\log n)$

② Non Comparision based Sorting algo:-

① Counting Sort

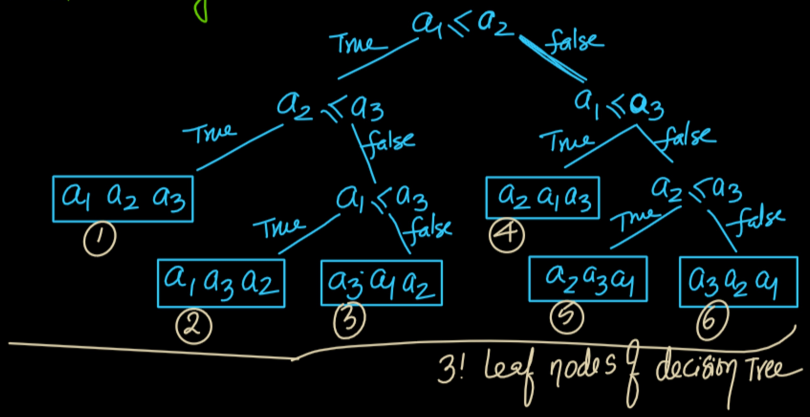
Time Complexity : $\Theta(n+k)$
Space Complexity : $\Theta(n+k)$
Stable Sorting algo but not inplace Sorting algo.

② Radix Sort

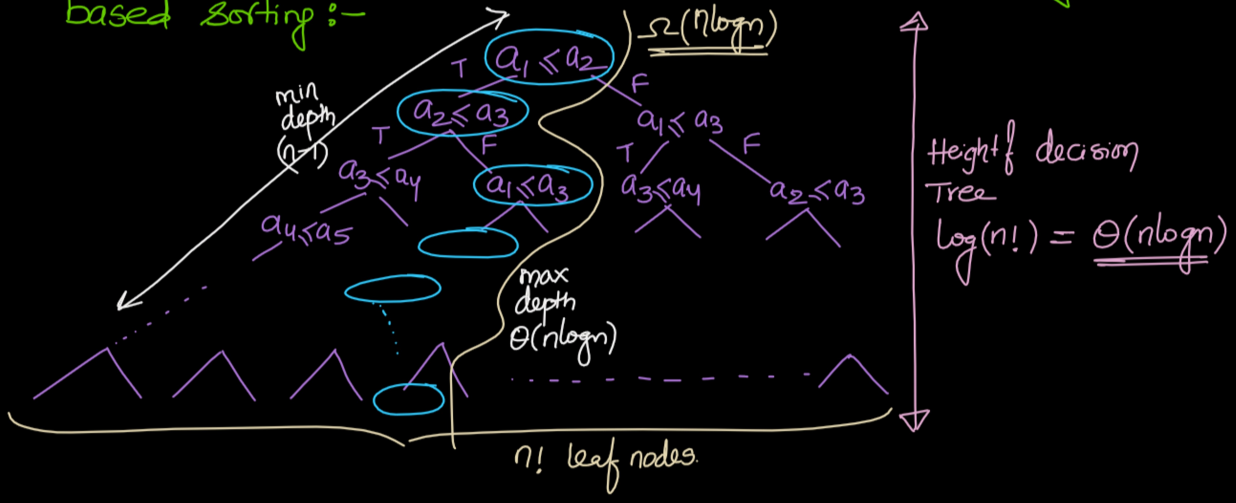
Time Complexity : $\Theta(nd)$
Space Complexity : $\Theta(n)$
Stable Sorting algo and inplace Sorting algo.

⇒ Maximum (Worst case) comparisions required to sort n elements using Comparision based Sorting algo is $\Omega(n \log n)$.

⇒ Decision Tree to sort 3 elements $[a_1, a_2, a_3]$ using Comparision based sorting :-



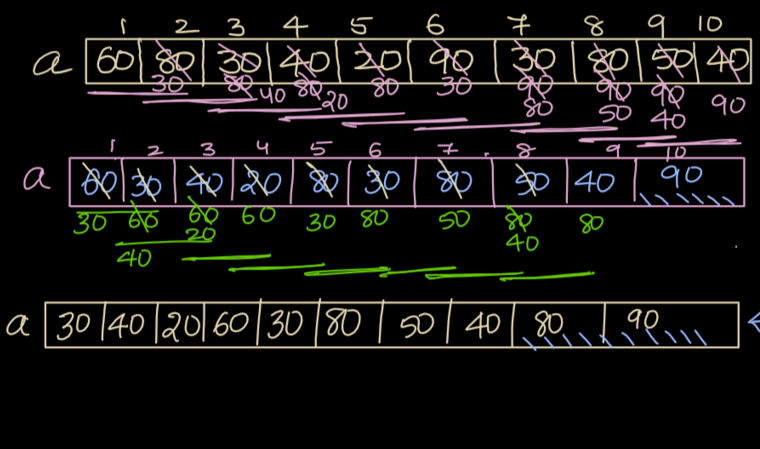
⇒ Decision Tree to sort n elements $\{a_1\ a_2\ a_3 \ldots a_n\}$ using Comparision based sorting :-



⇒ Worst case # of Comparisions to sort array of n elements using Comparision based Sorting $\Omega(n \log n)$.

⇒ Min # of Comparisions to sort array of n elements using Comparision based Sorting algo : $n-1$

Bubble Sort : Compare adjacency elements a[j], a[j+1]
if (a[j] > a[j+1]) swap(a[j], a[j+1]) Where j = 1 to n-i
and (i = 1, 2, 3, ··· n-1 passes.) // [Stable Sorting & Inplace Sorting]



Algo BubbleSort(a,n)
{
```
for(i=1; i<=n-1; i++)
{   flag = 0;
    for(j=1; j<=n-i; j++)
    {   if(a[j] > a[j+1])
        {   swap(a[j], a[j+1])
            flag = 1
        }
    }
    if(flag == 0) return
}
```

Time Complexity :- [AC & WC]

pass	#f Comp	#f Swap's
1	n-1	0 to (n-1)
2	n-2	0 to (n-2)
3	(n-3)	0 to (n-3)
4	⋮	⋮
⋮	⋮	⋮
n-1	1	0 to 1

$\frac{n(n-1)}{2} + \left\{0 \text{ to } \frac{n(n-1)}{2}\right\} = \Theta(n^2)$
{WC/AC} TC

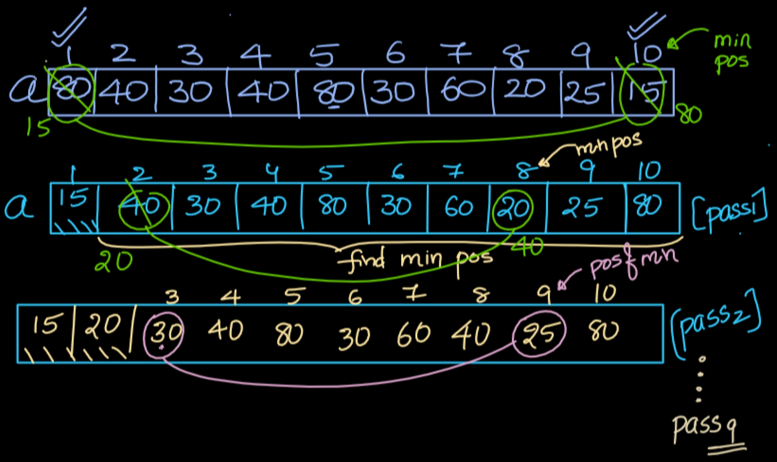
Best Case TC
#f comp
(n-1) Comp in 1st pass
$(n-1) = \Theta(n)$
{BC} TC

Selection Sort :-

Find position of min from a[i] to a[n] and swap with a[i]

where $i = 1, 2, 3, \ldots n-1$ passes.

[Selection Sort is inplace Sorting but not stable sorting algo]




```
a[1...n]
mp = 1
for(j = 2 to n)
{ if(a[j] < a[mp])
      mp = j
}
```

```
Algo SelectionSort(a,n)
{
   for(i=1; i<=n-1; i++)
   { // find pos of min from a[i] to a[n]
     minPos = i
     for(j = i+1; j <= n; j++)
     {    if (a[j] < a[minPos])
              minPos = j
     }
     Swap(a[i], a[minPos]);
   }
}
```

Time Complexity :-

Passes	#f Comp	#f Swaps
1	$(n-1)$	1
2	$(n-2)$	1
3	$(n-3)$	1
⋮	⋮	⋮
$n-1$	1	1

Time Complexity $\Rightarrow \frac{n(n-1)}{2} + (n-1) = \Theta(n^2)$ [for all cases]

// Selection Sort better

⇒ Time Complexity of Selection Sort : $\Theta(n^2)$ in all cases.

⇒ Only $(n-1)$ Swap's required to sort array.

⇒ Selection Sort Special case of Quick Sort with pivot element selected as min element.

Posterisy Analysis: Less elements to sort : Selection Sort Runs faster than QS.
More elements to sort: QS runs faster than SS.

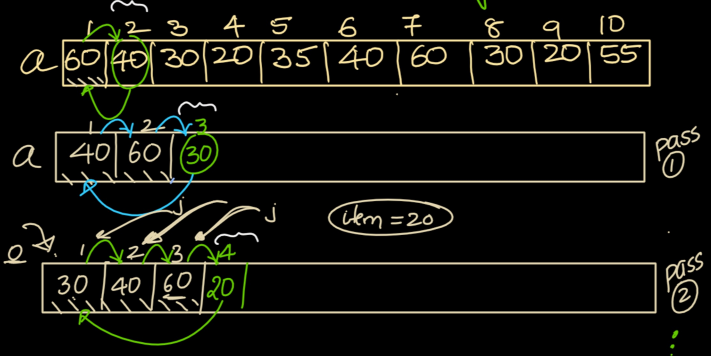
```
Sort (a, n)
{
    if (n<20) // small size array.
        SelectionSort (a,n)
    else
        QuickSort (a, n)
}
```

Insertion Sort:- Insert element a[i+1] into correct position of sorted part of array a[1] to a[i]. Where i = 1, 2, 3, .. n-1 (passes)

pass (i)

Sorted part of array.

After pass i
(i+1) elements in sorted order.




```
Algo InsertionSort (a,n)
{   for (i=1; i<=n-1; i++)
    {   // a[i+1] place into sorted part of array
        // a[1] to a[i]
        item = a[i+1] ; j = i
        while (j >= 1 and a[j] > item)
        {   a[j+1] = a[j]
            j-- ;
        }
        a[j+1] = item
    }
}
```

array shift

element comp.

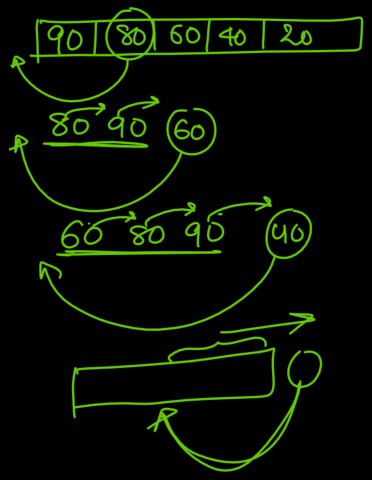
Time Complexity :-

passes	min Comp	min shift	Max Comp	Max arrayshift	Avg Comp	Avg arrayShift
1	1	0	1	1	1	1
2	1	0	2	2	2/2	2/2
3	1	0	3	3	3/2	3/2
⋮	⋮		⋮			
n-1	1	0	n-1	n-1	(n-1)/2	(n-1)/2

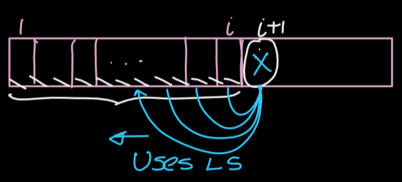
(n-1) Comp
BC TC of IS : $\Theta(n)$

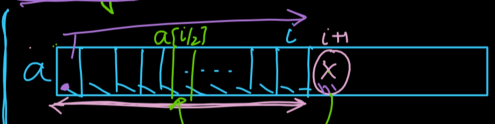
$\frac{n(n-1)}{2} + \frac{n(n-1)}{2}$
Worst Case TC of Insertion Sort : $\Theta(n^2)$

$\approx \frac{n(n-1)}{4} + \frac{n(n-1)}{4}$
Avg Case TC of Insertion Sort : $\Theta(n^2)$



⇒ Insertion Sort WIC TC $\Theta(n^2)$ uses linear search to find position of inserting element into sorted part of array. If Binary search uses to find position of inserting element then WIC TC of modified Insertion Sort?
a) remain $\Theta(n^2)$ b) becomes $\Theta(n\log n)$ c) becomes $\Theta(n)$ d) becomes $\Theta(\log n)$.





BS to find pos
& ($\log i$ Comp)
⇒ i array shift

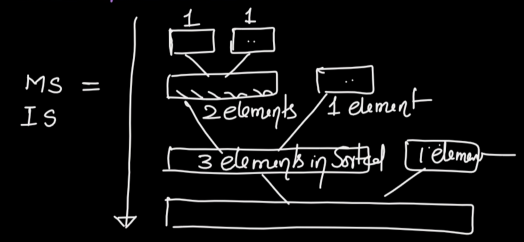
pass (i)

	Worst Case Comparision	Array shift Worst Case
pass1	1	1
pass2	log 2	2
pass3	log 3	3
⋮	⋮	⋮
pass n-1	log (n-1)	n-1

$\log(n-1)!$ + $\frac{n(n-1)}{2} = \Theta(n^2)$ array shift operation
$\approx \Theta(n\log n)$ Comparision

$\Theta(n^2)$

⇒ Insertion Sort special case of Merge Sort if 2nd sorted list for merge is always one element.



Insertion Sort is stable sorting algo && Inplace Sorting algo.

Posteriory Analysis:-

* for small array to perform sort
  Insertion Sort runs faster than Merge Sort.
* for large array to perform sort
  Merge Sort runs faster than Inserting Sort.

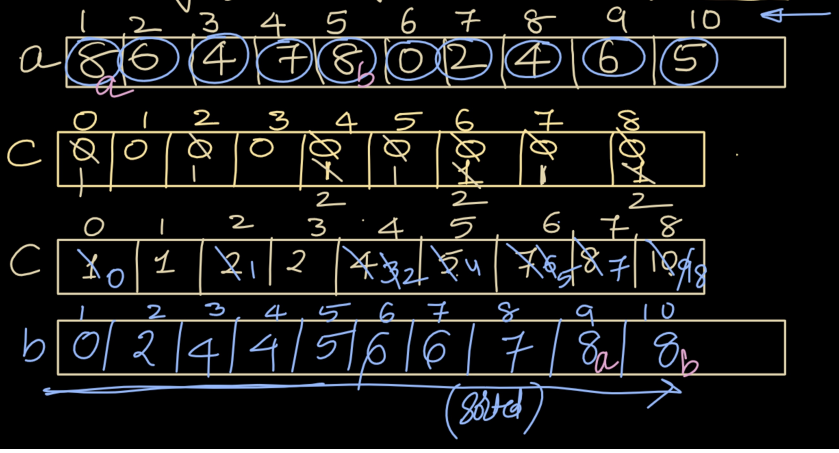
```
Algo Sorting(a,n)
{   if(n<20)
        Call InsertionSort(a,n)
    else
        Call MergeSort(a,n)
}
```

Non Comparision based Sorting Algo:-

① Counting Sort :- Given array of n integers and range of elements [0,K].

```
for(i=0; i<=K; i++)
    Ci = 0
for(i=1; i<=n; i++)
    C[a[i]]++;
for(i=1; i<=K; i++)
    Ci = Ci-1 + Ci
for(i=n; i>=1; i--)
{   b[C[a[i]]] = ai
    C[a[i]]--
}
```

Array of 10 integers elements range [0..8]



```
Algo CountingSort (a, n, k)
{ // A[1..n] array of n integers && Range of elements [0, k]
θ(k) { for( i=0 ; i≤k ; i++)
           c[i]=0
θ(n) { for( i=1 ; i≤n ; i++)
           c[a[i]]++
θ(k) { for(i=1; i≤k ; i++)
           c[i] = c[i-1] + c[i]
θ(n) { for(i=n; i≥1 ; i--)
     {     b[c[a[i]]] = a[i]
           c[a[i]]--
     }
return b[1..n]
}
```

⇒ Counting Sort TC: $\theta(n+k)$.

⇒ Best case TC of Counting sorting: $\theta(n)$ {if $k \leq n$}

⇒ SC of Counting Sort: $\theta(n+k)$
C[0..k] && b[1..n] arrays used

⇒ Counting sorting stable sorting but not inplace sorting.

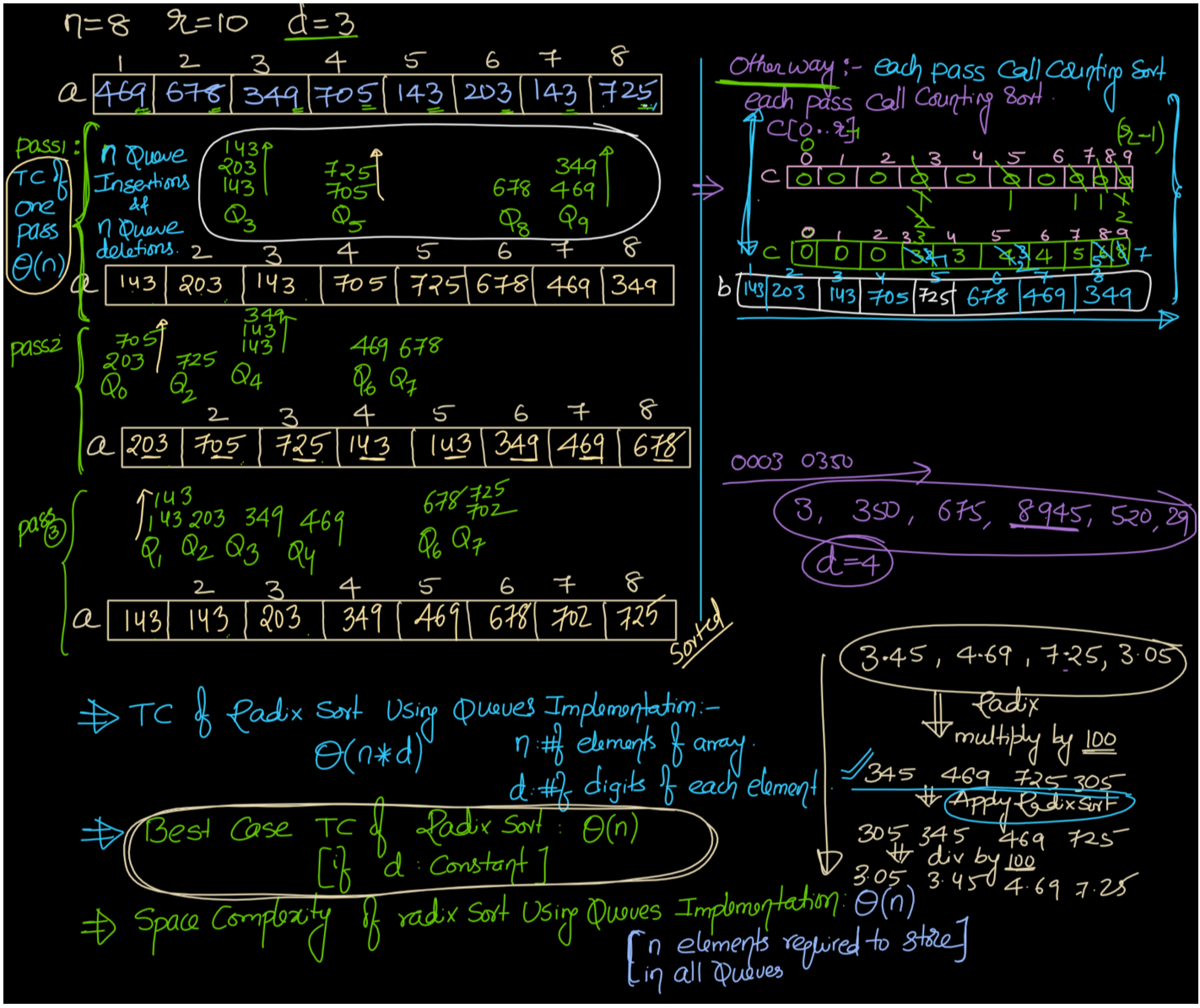
Radix Sort:- [bucket Sort].

radix[r] :- r different symbols used to represent any number whose values are 0, 1, 2, 3, ... r-1.

- $r = 10 \Rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $r = 16 \Rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ // [16 symbols]
- $r = 20 \Rightarrow \{0, 1, \cdots 9, A, B, C, D, E, F, G, H, I, J\}$ // 20 symbols.

Radix Sort:- Given n integers && each integer d digits, with radix - r.

Passi: Each element of array place in Queue-0 to Queue r-1 based on i^{th} least significant digit. (and) Retrieve all elements from Queue-0 to Queue r-1 and store elements into array.
Where $i = 1, 2, 3, \cdots d$ [passes]



Q] Using which algo an array of n elements in range [1..n^3] will be sorted using O(n) time ?

a) MergeSort $\Theta(n\log n)$ b) Quick Sort $\Theta(n^2)/\Theta(n\log n)$ c) Radix Sort d) Insertion Sort $\Theta(n^2)$ e) Counting Sort $\Theta(n^3)$ $\Theta(n+k)$

⇒ How many digits for element $X=(n^3)$ in decimal format (r=10)

# of decimal digits to represent $X=n^3$ in decimal (r=10) $\approx \log_{10}(x) = \log_{10}(n^3) = \lceil 3\cdot\log_{10} n \rceil$

{ # of digits to store element X in radix-r $= \lceil \log_r x \rceil + 1$ digits }

Method 1

Array of n elements range [1..n^3]

TC to sort by Using Radix Sort $= \Theta(n*d) = \Theta(n\log n)$

$d = \lceil \log_{10} n^3 \rceil$ digits.

Method 2

Given array of n elements whose range [1..n^3]

Step1] Convert each element of array into radix-n number system. $\Theta(n)$

element $x = (n^3)_{10}$

# of digits required to convert x into radix-n number system $= \log_n x = \lceil \log_n n^3 \rceil = 3$ digits (Const)

Step2] Apply Radix Sort for step① Resulted array. $\Theta(n)$

// Sorted array in result of Radix Sort.

Step3] Convert each element of Sorted array of radix-n to radix-10 [Given format]. $\Theta(n)$

TC: Sort array: $\Theta(n)$

Q] [MCQ] If Radix Sort to Sort n integers in the range $[n^{K/2}, n^K]$ for some $K > 0$ which is independent of n. Then time taken would be

a) $\Theta(n)$ b) $\Theta(K \cdot n)$ c) $\Theta(n \log n)$ d) $\Theta(n^2)$.

$[n^0, n^K]$

⇒ Direct Radix Sort for n elements range $[n^0 - n^K]$

$X = n^K$

(d) #digits for each element $= \log_{10} n^K = \lceil K \cdot \log_{10} n \rceil$

TC of Radix Sort: $\Theta(n \cdot d)$

$: \Theta(n \cdot K \log n)$

⇒ Convert elements into Radix-n number System

$X = n^K$ [element]

#digits for X in radix-n $\} = \lceil \log_n n^K \rceil = K$ digit

Apply Radix Sort

TC: $\Theta(n \cdot d) = \Theta(nK)$