Ex:- Calculate Time & space Complexity for the given Algo.

```
Algo: TOH(n,x,y,z) ——————> T(n)
      {
        if(n>=1) ———————— 1
        {
          TOH(n-1,x,z,y); —— T(n-1)
          Move x->z; ———————— 1
          TOH(n-1,y,x,z); —— T(n-1)
        }
      }
```

Time Complexity:

first find recurrence relation for Executed statement

$$T(n) = \begin{cases} 2T(n-1)+1, & \text{if } n>0 \\ 1, & \text{if } n=0 \end{cases}$$

Solution of Recurrence Relation $T(n) = 2T(n-1)+1$

By Apply master method for Decreasing function

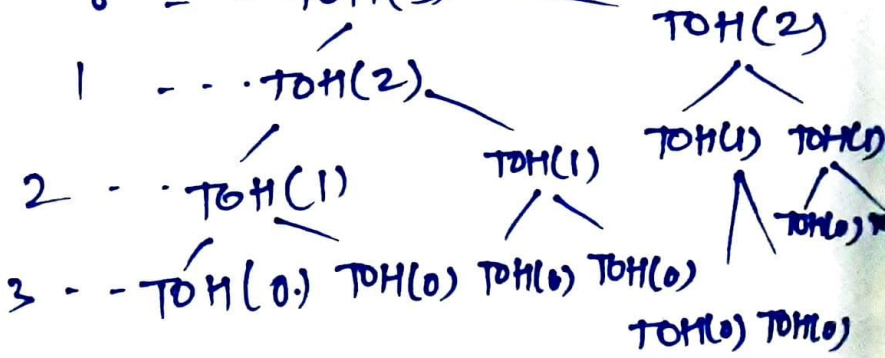
$$T(n) = 2T(n-1)+1 = O(2^n)$$

$$\# TC = O(2^n)$$

Space Complexity:

for space complexity find STACK space

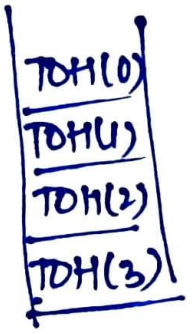
let $n = 3$



Stackspace:

No. of function in STACK = 4 = n+1 or height+1

$$\# SC = O(n)$$



Divide & Conquer :-

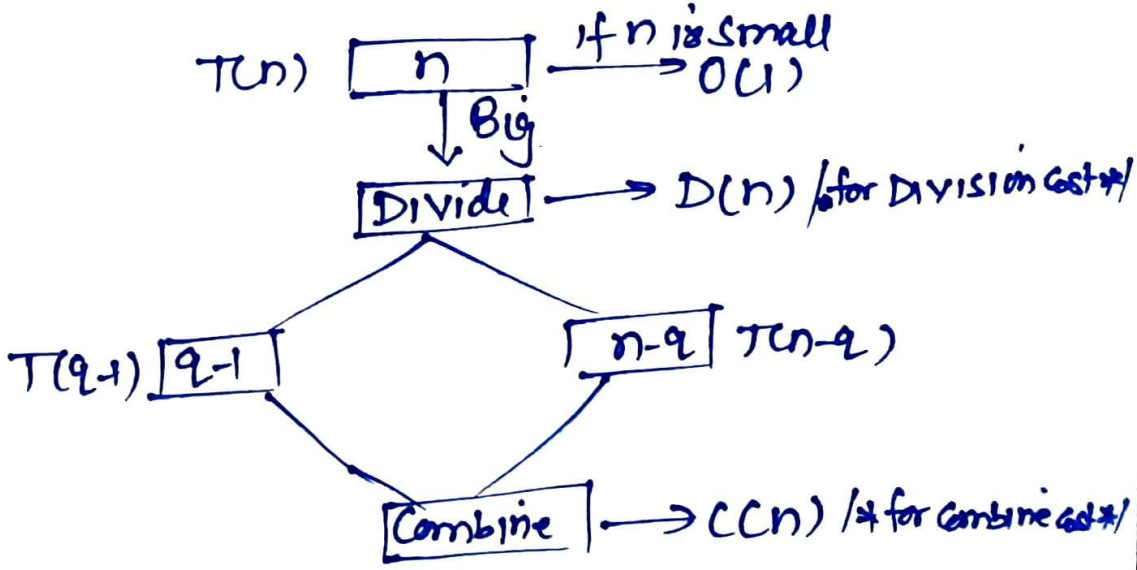
1- **Divide :-** Divide the big problem into small problem

A[P, r] → A[P.... q], A[q+1.... r]

2- **Conquer :** The two subarrays Sub problem by calling recursively until Sub Problem solved.

3- **Combine :-** The Sub problem solved so that we will get find Problem Solution.

```
Algo: DAC(P, r)
{ if (small(P,r)),
    return (Solution(P,r));
  else
  { q = Divide(P,r);
    a = DAC(P,q);
    b = DAC(q+1,r);
    c = Combine(a,b);
    return c;
  }
}
```



Recurrence Relation :-

$$T(n) = \begin{cases} O(1) & \text{if } n \text{ is small} \\ T(q-1) + T(n-q) + D(n) + C(n) & \text{if } n \text{ is big} \end{cases}$$

Binary Search:- Array should be Sorted.

Step1: First find the middle element of the array

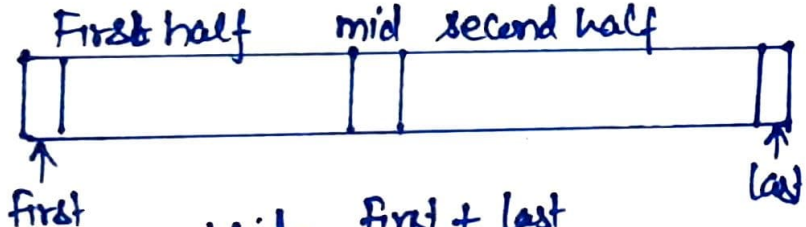
Step2: Compare the mid element with an item

Step3: There are three cases:

i) if it is a Desired element Then Search Successful.

ii) If it is less than desired item then search only the 2nd half of the array.

iii) if it is greater than the desired element Then search only the ~~first~~ 1st half of the array.



$$Mid = \frac{First + Last}{2}$$

Recurrence Relation for Binary search:

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

Solution of Recurrence Relation $T(n)=T(n/2)+1$

$= O(\log_2 n)$ #TC: BC=O(1)

AC=WC=$O(\log_2 n)$

$T(n)$

Algo: Binary search(a, i, J, x)

```
{ int mid;
  If (i==J)  /* for small problem */
  { if (a[i]==x)
      return(i);
    else
      return(-1);
  }
  else
  { mid = (i+J)/2;
    If (a[mid]==x)
      return(mid);
    else if (a[mid]>x)
      Binary search(a,i,mid-1,x);
    else
      Binary search(a,mid+1,J,x);
  }
}
```

O(1) — the If (i==J) block

O(1) — mid = (i+J)/2;

O(1) — If (a[mid]==x) return(mid);

$T(n/2)$ — Binary search(a,i,mid-1,x);

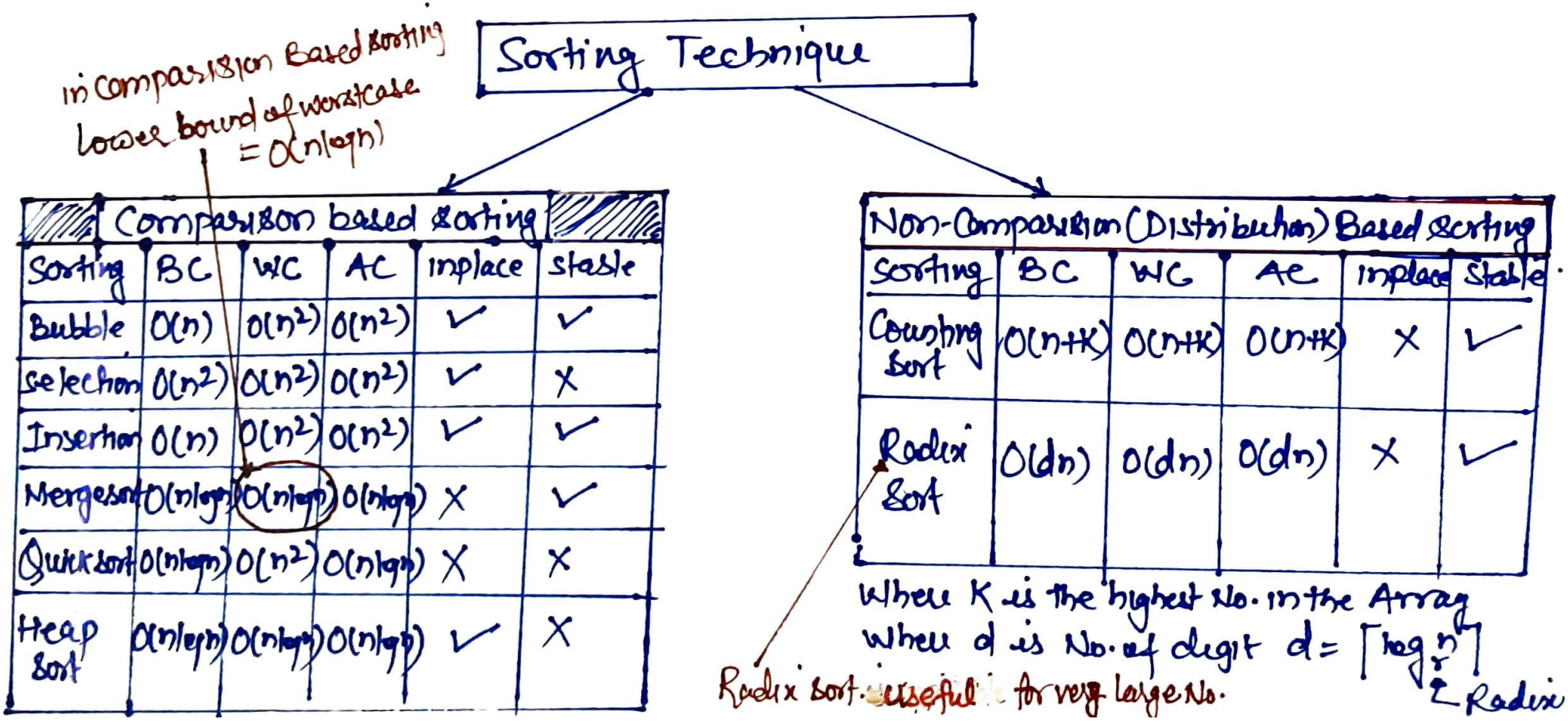
$T(n/2)$ — Binary search(a,mid+1,J,x);

Sorting:- Arranging elements in a specific order either in ascending or in descending order is know as sorting.

inplacesort:- Extra Space required = O(1) or O(logn), Ex:- Bubble, Selection, Insertion, Heapsort etc.

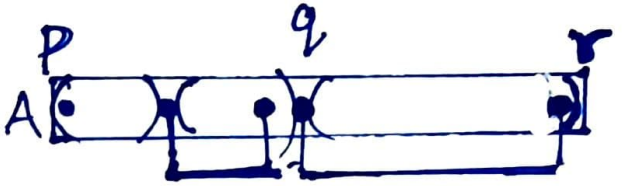
Stablesort:- The order of the repeated elements is not changed after sorting, Then that sorting is called Stable sort., Ex:- Bubble, Insertion, mergesort, counting sort, Radix sort etc.

Ex:- 5 (2) 4 6 [2] 1 3, After Sorting = 1 (2) [2] 3 4 5 6 (Stable sort)



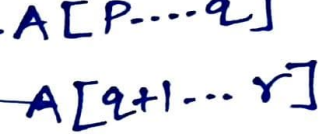
Quick Sort:-

- Based on Divide & Conquer Paradigm.
- not inplace & not Stable sort
- Most Practically used Algorithm
- Consumes relatively fewer resources during execution.



Divide & Conquer:

1) **Divide:** Divide The Large array into small arrays $A[P,r]$ such that $A[P \ldots q] \le A[q+1 \ldots r]$



2) **Conquer:** The two subarrays $A[P \ldots q]$ and $A[q+1 \ldots r]$ are sorted by recursive calls to quicksort.

3) **Combine:** Since the subarrays are sorted inplace no work is needed to combine them the entire array $A[P \ldots r]$ is now sorted

NOTE:- Quick sort uses Partition Algo. in which Pivot element is the first element or the last element.

Algo: Quicksort(A,P,r) — T(n)

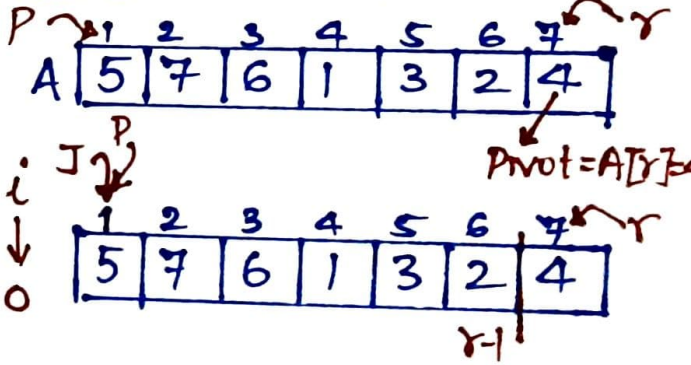
```
{ if (P<r)          —— O(1)
  { q = Partition(A,P,r); —O(n)
    Quicksort(A,P,q-1); —T(q-1)
    Quicksort(A,q+1,r); —T(n-q)
  }
}
```

Recurrence relation:

$$T(n) = \begin{cases} O(1), & \text{if } n=1 \\ T(q-1)+T(n-q)+n, & \text{if } n>1 \end{cases}$$

Algo: Partition(A,P,r)

```
{ x = A[r];
  i = P-1;
  for(J=P to r-1)
  { if (A[J] ≤ x)
    { i = i+1;
      Exchange A[i] ↔ A[J]
    }
  }
  Exchange A[i+1] ↔ A[r];
```



#TC : of partition Algo.
BC = WC = AC = O(n)

Space cost: = I/p cost + Extra stack space (O(1))

BC AC WC

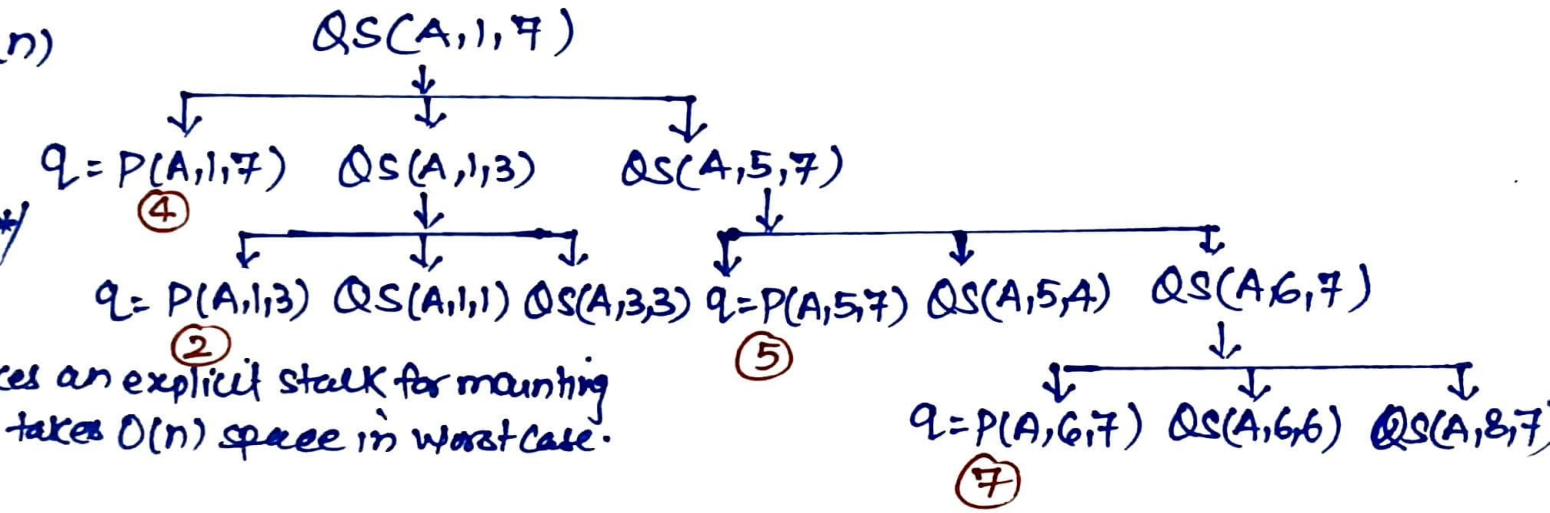
O(logn) O(logn) O(n)

A	5	7	6	1	3	2	4
	1	2	3	4	5	6	7

p → 1 , r → 7

#SC: BC = AC = O(logn)

WC = O(n) /*not inplace*/



NOTE: iterative implementation takes an explicit stack for maintaining call history so that Quicksort takes O(n) space in worst case.

Time cost:

Recurrence Relation: $T(n) = T(q-1) + T(n-q) + n$, if $n > 1$

Best case: If $q = n/2$, $n > 1 \Rightarrow T(n) = T(n/2 - 1) + T(n - n/2) + n$

$\Rightarrow \boxed{T(n) = 2T(n/2) + n}$, Apply master Method $= O(n \log n)$

$\boxed{BC = O(n \log n)}$

Worst case: If $q = 1$ (if list already sorted or all keys same)

$T(n) = T(0) + T(n-1) + n$, if $n > 1$,

$\boxed{T(n) = T(n-1) + n} = O(n^2)$, $\boxed{\text{Worst case} = O(n^2)}$

$\boxed{T(n) = T(c) + T(n - (c+1)) + n}$ where c is constant $c \geq 0$

Avg case:

$\boxed{T(n) = T(n/3) + T(2n/3) + n}$, $\boxed{= O(n \log_{3/2} n) = O(n \log n)}$

$\boxed{T(n) = T(\alpha n) + T((1-\alpha)n) + n}$ where α = any fraction like $1/3, 1/5, 1/10, 1/9$ etc $(0 < \alpha < 1) = O(n \log_{1/(1-\alpha)} n)$

Ex:- using Quicksort to sort in Ascending order

t_1 time to sort = 1,2,3,4,5 ..... n

t_2 time to sort = n, n-1, n-2, ... 3,2,1

TRUE:

a) $t_1 = t_2$ b) $t_1 > t_2$ c) $t_2 > t_1$ d) None

Ex:- A machine needs a minimum of 100 sec to sort 1000 names by Quicksort the minimum time needed to sort 100 names will be Approx.

a) 50.2 sec

b) 6.7 sec

c) 72.7 sec

d) 11.2 sec

$$\frac{100}{t_2} = \frac{1000 \log 1000}{100 \log 100} = 6.7 \text{ sec.}$$

Ex:- In Quicksort the sorting of n no. the $n/3^{th}$ largest element is selected as Pivot using $O(n^3)$ Complexity Algo. What will be Worst Case time Complexity?

a) O(nlogn)

b) $O(n^2)$

c) O(n)

d) $O(n^3)$

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

$$T(n) = O(n^3) + O(n \log n)$$

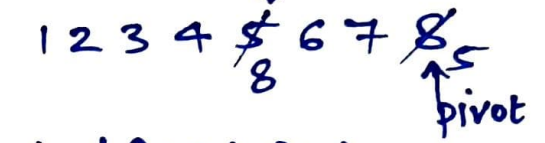
$$= O(n^3)$$

Ex:- In Quicksort the sorting n elements Pivot is $(n/4)^{th}$ element selected using O(n) time Algo. Then. What is the Worst Case time Complexity of Quicksort.

$$T(n) = O(n) + T(n/4) + T(3n/4) + n$$

$$= O(n) + O(n \log n) = O(n \log n) \text{ Ans}$$

Randomized Quicksort:



Algo:- Randomized QS(A, P, r)

```
{
  if (P<r)
  { i ← random(P, r);
    swap(A[i], A[r]);
    q ← partition(A, P, r);
    Randomized QS(A, P, q-1);
    Randomized QS(A, q+1, r);
  }
}
```

NOTE:

Expected Worst case Time = O(nlogn)

But still Worst case Time = $O(n^2)$

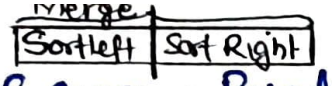
Ex: A

2	2	2	2	2	2	2

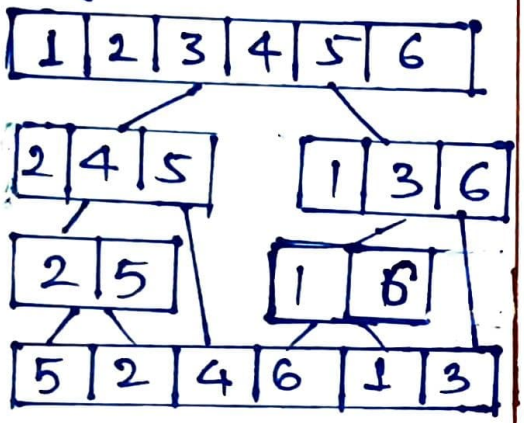
Apply randomized Quicksort

Worst case = $O(n^2)$

Mergesort:



- Follow Divide & Conquer Paradigm.
- It is Stable sort
- It is not inplace sort.
- Time Complexity
  BC=WC=AC=O(nlogn)
- Space Complexity:
  BC=WC=AC=O(n)



In merge sort Algo:.
- We divide the array until array size is one
- When the sequence to be sorted has length 1 There is no work to be done i.e every sequence of length 1 is already in sorted order.
- Key opelation of this Algo. is the merging of two sorted sequence in the combine step
- To perform merging, we use an auxiliary Procedure

  Merge(A,P,q,r)
  A: an array
  p: index of first element in an array
  q: index of middle element " "
  r: index of last element " "

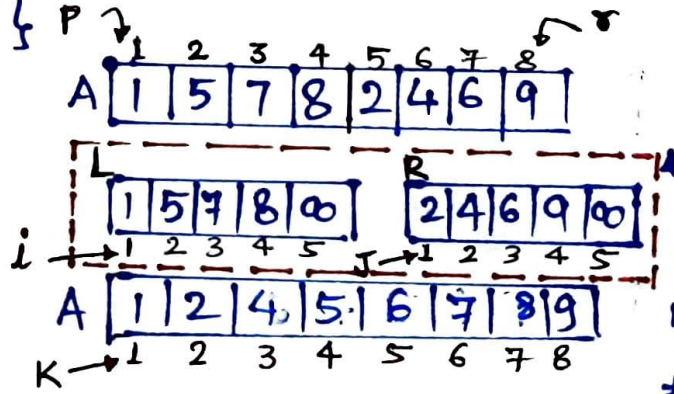
The merge( ) procedure assumes that the sub arrays A[P...q] & A[q+1.....r] are in sorted order. it merge Them to form a single sorted Array A[P.....r].

Algo: Merge(A,P,q,r)

```
{ n1 = q-P+1
  n2 = r-q
  Let L[1....n1+1] and R[1 to....n2+1] be new arrays
  for (i=1 to n1)
     L[i] = A[P+i-1]
  for (J=1 to n2)
     R[J] = A[q+J]
  L[n1+1] = ∞
  R[n2+1] = ∞
  for (K=P to r)
     if (L[i] ≤ R[J])
          A[K] = L[i++]
     else
          A[K] = R[J++]
}
```

NOTE: list A has m elements, list B has n elements
Total Comparision: m+n-1 = O(m+n)
Total Movement = O(m+n)
Total Time for Merge = O(m+n)

Ex:-



Total Extra Space = O(n)

#SC:.
BC=AC=WC=O(n)

#TC:.
BC=WC=AC=O(n)

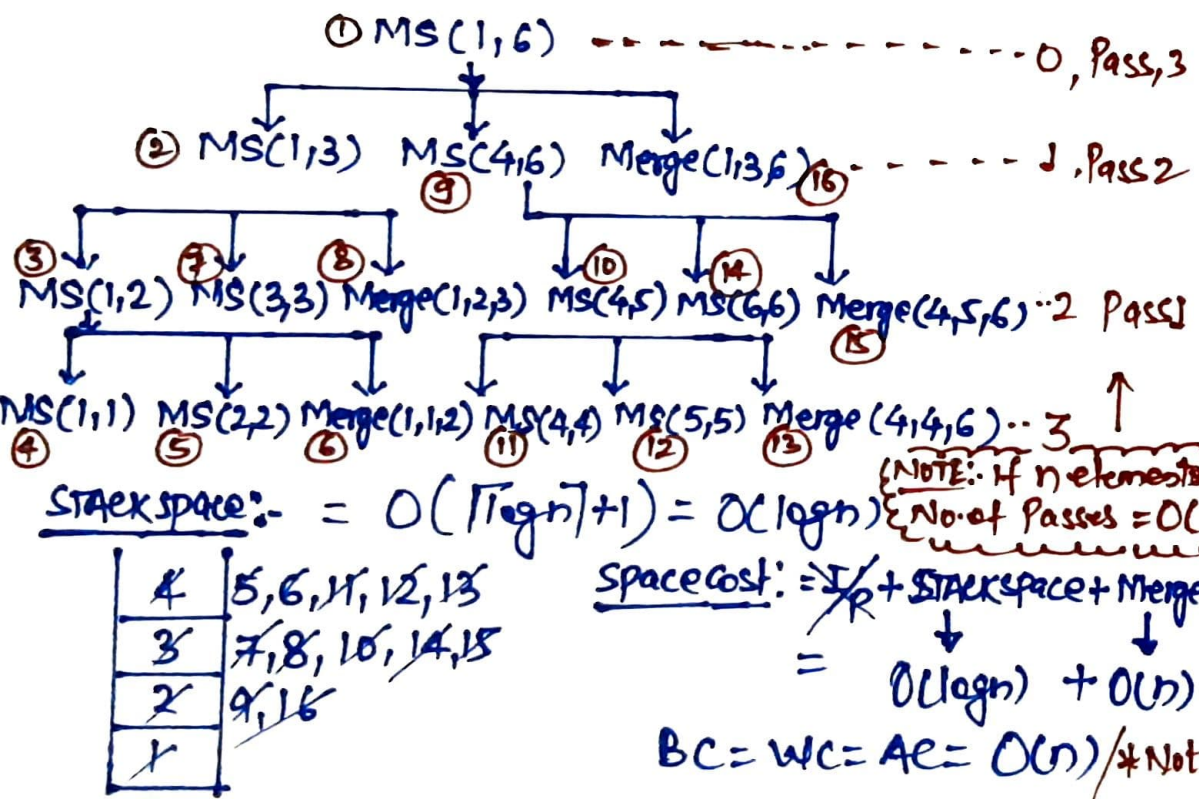
#TC:. = Comparision + Movement
= (n/2+n/2-1)+(n/2+n/2) = O(n):

```
Algo: Mergesort (A,P,r) ---- T(n)
{ if (P<r)              ---- O(1)
  { q = ⌊(P+r)/2⌋       ---- O(1)
    Mergesort (A,P,q)   -- T(n/2)
    Mergesort (A,q+1,r) -- T(n/2)
    Merge (A,P,q,r)     -- O(n)
  }
}
```

Ex:-

	1	2	3	4	5	6
A	5	2	4	6	1	3

P → 1, r → 6



Stack space:- $= O(\lceil \log n \rceil + 1) = O(\log n)$

4	5, 6, 11, 12, 13
3	7, 8, 10, 14, 15
2	9, 16
1	

(NOTE: If n elements Then No. of Passes = $O(\log n)$)

Space cost: = I/P + Stackspace + Mergespace

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

BC = WC = AC = $O(n)$ /* Not inplace */

Recurrence Relation:

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

Time Complexity: $T(n) = 2T(n/2) + n$, Apply Master Method

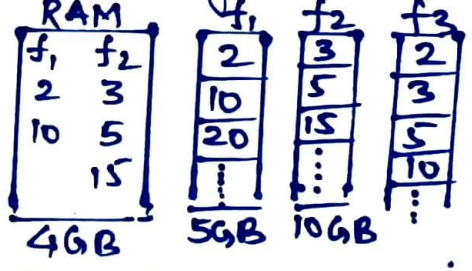
$$BC = WC = AC = O(n \log n)$$

NOTE:- Mergesort has the Best Worstcase behavioral but require More Storage Than heapsort

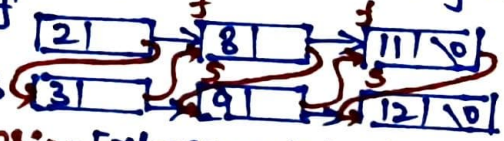
Pros & Cons of Mergesort:

Pros:-

- Use for large size list



- Support External sorting
- Suitable for Linked list sorting b/c No. Extra space required



Mergesort and Insertion sort both are Suitable for Linked list sorting

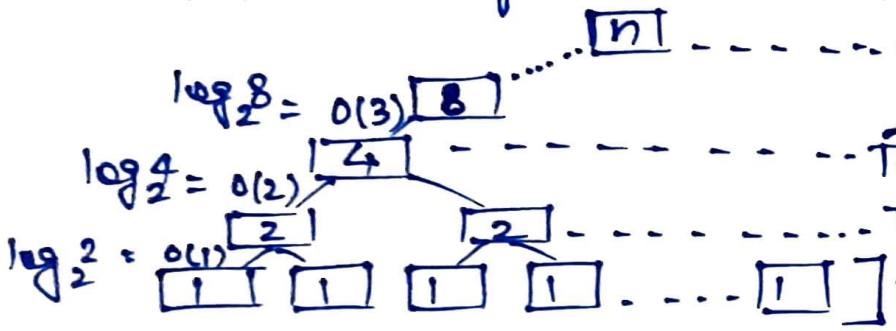
Cons:-

- Extra space (not inplace) required not suitable for Arrays
- not suitable for Small problem
- Recursion overhead.

Ex:- Straight two way merge sort Algo. to sort the following elements in Ascending order
20,47,15,8,9,4,40,30,12,17 Then the order of these elements after second Pass of the Algo.

a) 8,9,15,20,47,4,12,17,30,40
b) 8,15,20,47,4,9,30,40,12,17
c) 15,20,47,4,8,9,12,30,40,17
d) None

Ex:- Given n elements Merge them into one sorted list using Merge Procedure Total Cost=?



Total Merging cost = $O(n) \times O(\log_2 n) = O(n \log_2 n)$ — Any

Total Merging = $O(n) \times O(\log_2 4)$

Total Merging = $O(n) \times O(\log_2 2)$

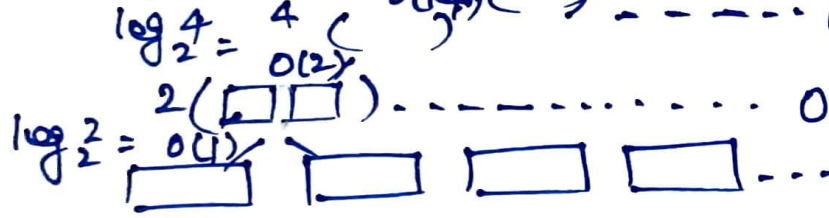
Total = n elements

Cost of total elements copy from one level to another = $O(n)$

NOTE:- To merge K-sorted lists into one sorted list, where n is the total number of elements in all the input list = $O(n \log K)$

Ex:- n strings each of length n are given them what is the time taken to sort them in Lexicographical order using Merge Sort

How much Time to Compare = $O(n)$



$O(n^2) \times O(\log n) = O(n^2 \log n)$ — Any

$O(n^2)$

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

Compare & Movement

Ex:- Assume the merge sort Algo. in the worst case takes 30 sec for an input of size 64 which of the following most closely approximately the maximum input size of a problem that can be solved in 6 minits. ?

time = $O(n \log n)$

$$\frac{30}{6 \times 60} = \frac{64 \log 64}{n \log n}$$

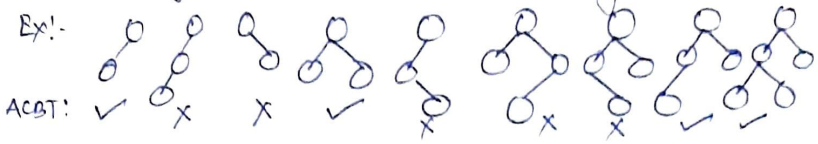
$n \log n = 4608 \Rightarrow n = 512$ Ans

a) 256 b) 512 c) 1024 d) 2048

Binary Heap:- is a Almost Complete binary tree with heap properties

Almost Complete Binary Tree (ACBT):- Filled from left to right

Ex:-

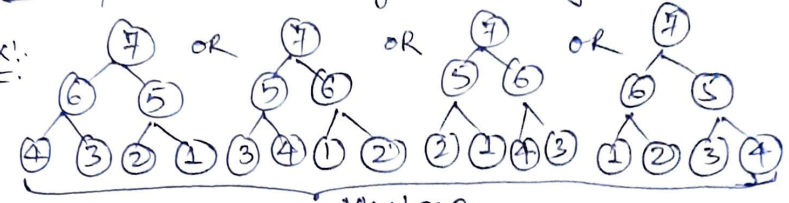


Heap properties:

i) **Maxheap:-** Parent ≥ left & Right child at every level

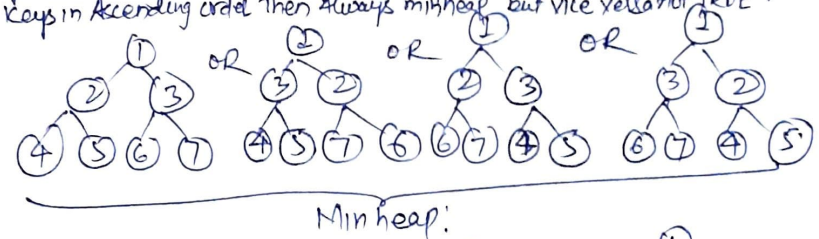
ii) **Min heap:-** Parent ≤ Left & Right child at every level.

Ex:-

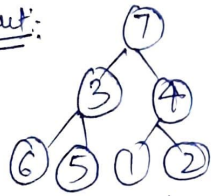


NOTE: 1) If Keys in decreasing order Then Always max heap But vice versa not TRUE.

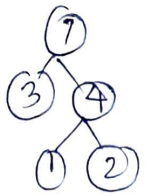
2) If Keys in Ascending order Then Always min heap but vice versa not TRUE.



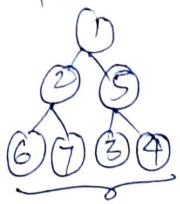
But:



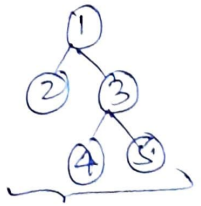
not max heap



not maxheap B/C not ACBT.



not min heap



not min heap B/C not ACBT.

Representation of heap:-

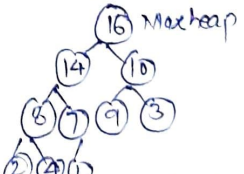
Algo: Parent(i)
- return $\lfloor i/2 \rfloor$;

Left(i)
- return $2i$;

Right(i)
- return $2i+1$;

A

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

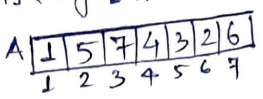


NOTE:
1) minimum element on height $h = 2^h$
2) maximum element on height $h = 2^{h+1} - 1$

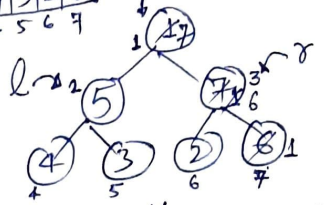
$$h+1 \le N \le 2^{h+1} - 1$$

Heapify:- To maintain heap property (max heap OR min heap)

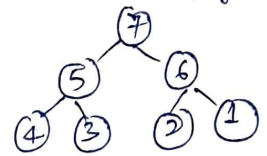
Heapsize[A] ← length[A]



Heapify (A, 1)



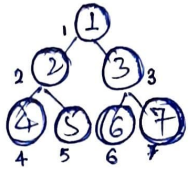
Heapify (A, 2)



Algo:. Heapify (A, i)

```
1 - l ← left(i)
2 - r ← right(i)
3 - if l ≤ heapsize[A] and A[l] > A[i]
4 -     Then largest ← l
5 - else  largest ← i
6 - if r ≤ heapsize[A] and A[r] > A[largest]
7 -     Then largest ← r
8 - if largest ≠ i
9 -     Then Exchange A[i] ↔ A[largest]
10 -        Heapify(A, largest);
```

NOTE:. for Construct heap call heapify from Bottom to up.



#sc: O(logn).
#TC: O(logn)

Running Time of Heapify:- for Worst case

Recurrence Relation $T(n) = T(\lfloor 2n/3 \rfloor) + O(1)$ $= O(\log n) = O(\log n)$

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

to fix up the relationship among the element A[i], A[Left(i)] and A[right(i)]

Build heap:- since the element in the subarray $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ are all leaves of the tree. each is a 1-element heap to begin with the Procedure Build heap goes through the remaining nodes of the tree and run heapify one each one

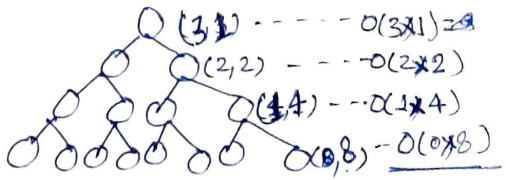
Algo: Buildheap(A)

```
1 - Heapsize[A] ← Length[A]
2 - For i ← ⌊Length[A]/2⌋ down to 1
3 -     Heapify(A, i)
```

Each call to heapify cost O(logn) Time and there are O(n) such calls Thus the running time is atmost O(nlogn) This is upper bound But not Asymptotically tight.

Tighter Analysis:-

No. of Nodes of Height $h = \left\lceil \frac{n}{2^{h+1}} \right\rceil$



The time required by heapify when called on a node of height h is $O(h)$

Total time taken will be max/min Heap $= \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \times O(h)$

$$= \frac{Cn}{2} \sum_{h=0}^{\log n} \left(\frac{h}{2^h} \right)$$

$$= O(n)$$

Build heap Cost in $\boxed{BC = WC = AC = O(n)}$.

Heapsort:-

Algo: Heapsort(A)

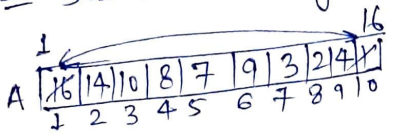
1 - Build heap(A) ---------- $O(n)$
2 - For $i \leftarrow$ Length[A] down to 2 ---------- $O(n-1)$
3 - do Exchange $A[1] \leftrightarrow A[i]$ ---- $O(n-1)$
4 - Heapsize[A] $\leftarrow$ Heapsize[A] - 1 ... $O(\log n)$
5 - Heapify(A,1) ---- $O((n-1) \times \log_2 n)$

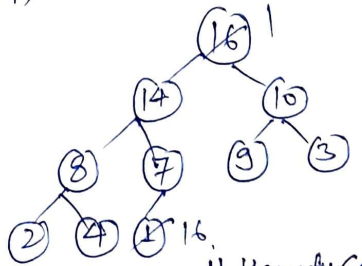
OR: $= \log n + \log(n-1) + \log(n-2) + \dots + \log 2 + \log 1$

$= \log (n \times (n-1) \times (n-2) \times \dots \times 2 \times 1)$

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

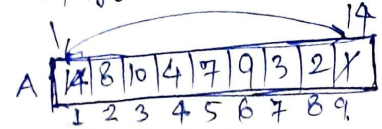
#TC: $BC = WC = AC = O(n \log n)$

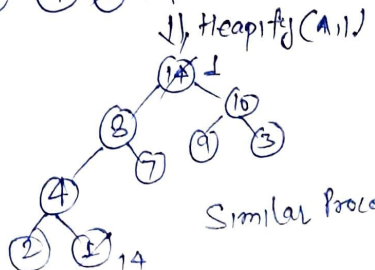




Heapify(A,1)

Heapify(A,2)

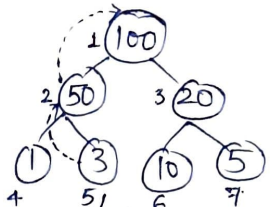




Similar Process!

Heap increase Key:-

```
Heapincrease(A, i, Key)
{ if (key < A[i])
    "Error"
  A[i] = key                    // Never go beyond root
  while (i>1 and A[i/2] < A[i])  // Parent of i
  { Exchange A[i] ↔ A[i/2];
    i = i/2;
  }
}
```



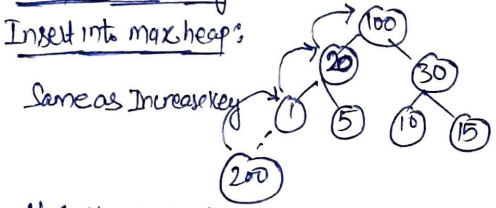
Increase 3 by 300

#TC: O(logn) < WC / AC, BC=O(1)

#SC: O(1)

Heap decrease Key: Similar to increase key

Insert into max heap:



Same as Increase key

#TC: BC = O(1), WC = AC = O(log n)

#SC: O(1)

#No of swap: AC = WC = O(log n), BC = O(1)

~~Algo: HeapExtractmax(A)~~

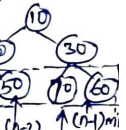
Extract max:

```
Algo: HeapExtractmax(A)
{ if (Heapsize[A] < 1)
    Error "Underflow";
  Max = A[1];
  A[1] ↔ A[Heapsize[A]]
  Heapsize[A] = Heapsize[A] - 1;
  MaxHeapify(A, 1);
  return max
}
```

#TC: O(logn)

#SC: O(logn)

NOTE:- from min heap:



1- Search minimum = O(1)

2- Search max from min heap = O(n), BC = O(1), AC = WC = O(n)

3- Search 7th max from min heap = O(n)

4- Search 7th min from min heap

$= 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 2^7 - 1 = 127$

Since 127 constant = O(1)

5- Delete last element = O(1)

6- Delete min from min heap = O(log n)

7- Delete n^{th} min (max) from min heap = O(n)

8- Delete $(n-1)^{th}$ min " " = O(n)

9- Delete $(n-10)^{th}$ min " " = O(n)

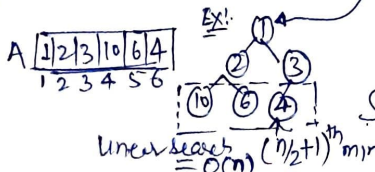
10- Delete $(n/2+1)^{th}$ min " " = O(n)

11- Delete max element = O(n)

12- Delete 150^{th} min " " = (150-1) log n = O(log n)

Extract min:-

Heap extract min Similar to Extract max:



= O(n)

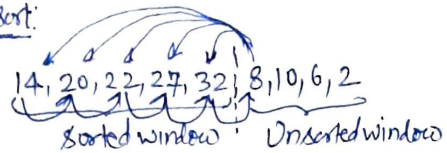
Similarly for max heap:

Shell sort: (Generalisation of Insertion sort)

Shell sort is a highly efficient sorting Algo. and is based on Insertion sort Algo. This algorithm avoids large shift as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

Ex:- 22, 27, 14, 20, 32, 8, 10, 6, 2

Insertion sort:



Large shifting operation in Insertion sort for elements 8, 10, 6, 2 because Adjacent elements Compared.

But in Shell sort Compare distance elements not near elements.

Shellsort:

- Based on Insertion sort
- Compare elements that are distant apart rather than adjacent
- Space b/w elements is Know as GAP/Interval
- Gap is $\lfloor n/2 \rfloor$, where n is No of elements in array
- In every Pass gap = $\lfloor gap/2 \rfloor$ till we reach last pass when gap is 1
- When gap is 1 Then Shell sort become insertion sort.
- Complexity: #TC: worstcase $\leq O(n^2)$

  BC = $O(n \log n)$

Algo:

```
for(gap = n/2; gap >= 1; gap = gap/2)
{ for(J=gap; J<n; J++)
  { for(i=J-gap; i>=0; i=i-gap)
    { if(a[i+gap] > a[i])
      { break;
      }
      else { swap(a[i+gap], a[i]);
           }
    }
  }
```

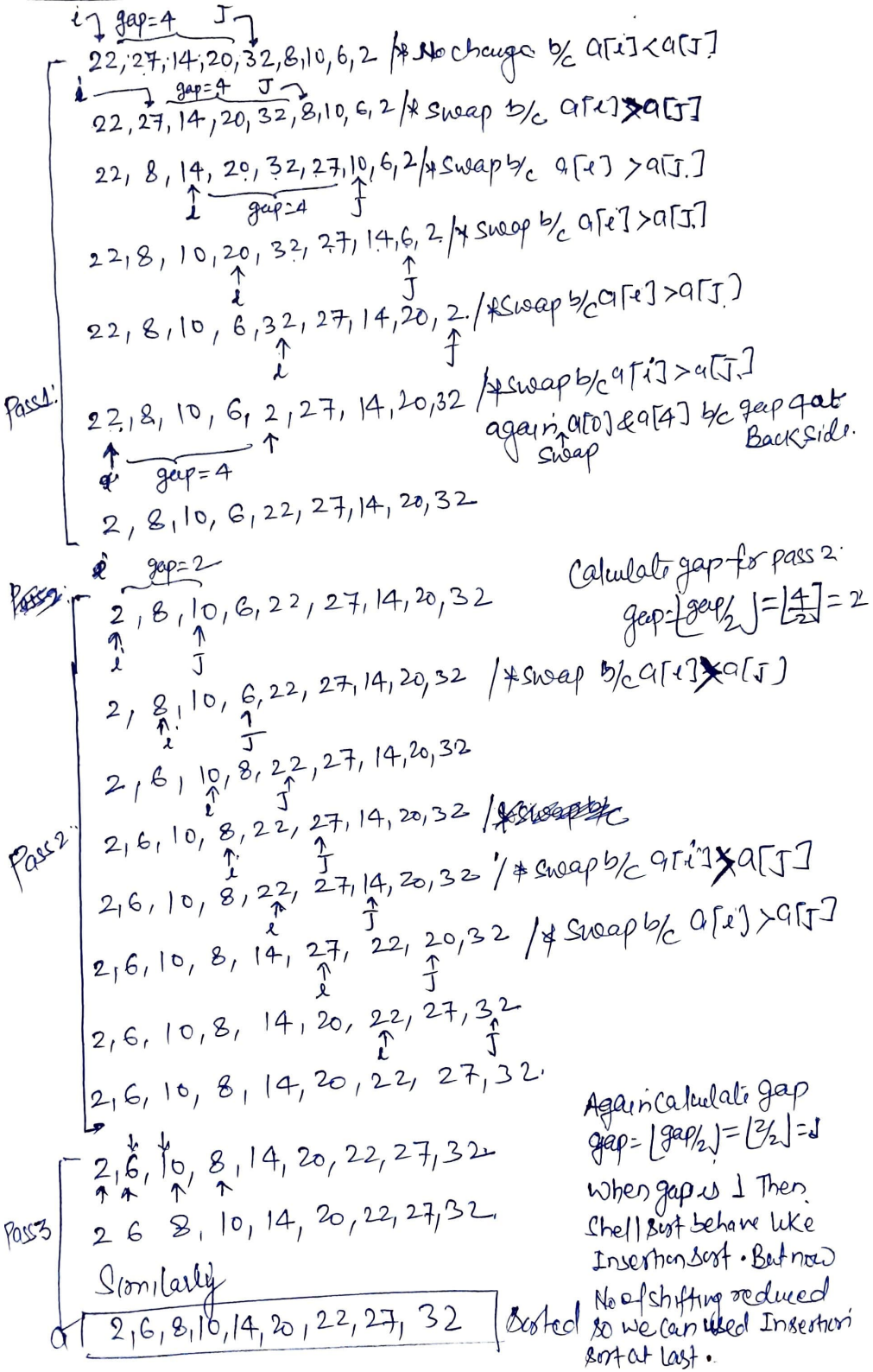
Ex: a

22	27	14	20	32	8	10	6	2
0	1	2	3	4	5	6	7	8

Given n=9, gap = $\lfloor n/2 \rfloor = \lfloor 9/2 \rfloor = 4$

~~Select pos~~

Goto next pos


gap=4
22,27,14,20,32,8,10,6,2 /* No change b/c a[i]<a[J]
22,27,14,20,32,8,10,6,2 /* swap b/c a[i]>a[J]
22,8,14,20,32,27,10,6,2 /* swap b/c a[i]>a[J]
22,8,10,20,32,27,14,6,2 /* swap b/c a[i]>a[J]
22,8,10,6,32,27,14,20,2 /* swap b/c a[i]>a[J]
Pass1:
22,8,10,6,2,27,14,20,32 /* swap b/c a[i]>a[J]
again a[0] & a[4] b/c gap 4 at Backside.
swap
gap=4
2,8,10,6,22,27,14,20,32
gap=2
Calculate gap for pass 2:
gap=⌊gap/2⌋=⌊4/2⌋=2
2,8,10,6,22,27,14,20,32
2,8,10,6,22,27,14,20,32 /* swap b/c a[i]>a[J]
2,6,10,8,22,27,14,20,32
Pass2:
2,6,10,8,22,27,14,20,32 /* swap b/c
2,6,10,8,22,27,14,20,32 /* swap b/c a[i]>a[J]
2,6,10,8,14,27,22,20,32 /* swap b/c a[i]>a[J]
2,6,10,8,14,20,22,27,32
2,6,10,8,14,20,22,27,32
2,6,10,8,14,20,22,27,32
Pass3
2 6 8 10 14 20 22 27 32
Again calculate gap
gap=⌊gap/2⌋=⌊2/2⌋=1
When gap is 1 Then Shell Sort behave like Insertion Sort. But now No of shifting reduced so we can used Insertion sort at last.
Similarly
2,6,8,10,14,20,22,27,32
Sorted

Sorting By Distribution: Linear time sort [O(n)]

1) Counting sort
2) Radix sort
3) Bucket sort

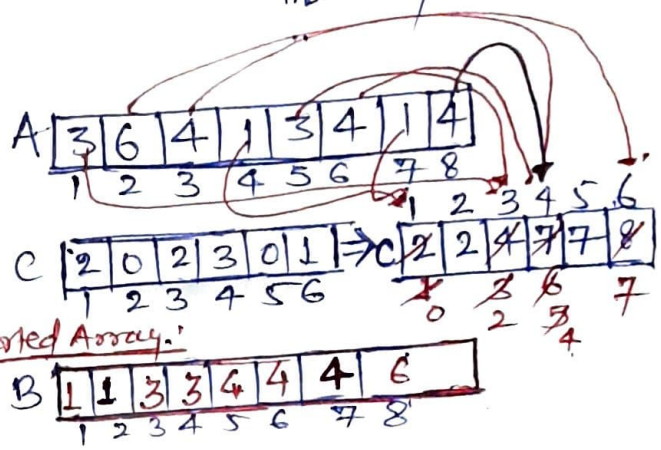
Counting Sort: - Counting sort Assumes that each of the n-input elements is an integer in the range 1 to K, for some integer $K \approx O(n)$

The Basic idea of Counting sort is to determine, for each element input element x, the no. of elements less than x. This information can be used to place element x directly into its position in the O/p array, If there are 15 elements less than x, Then x belongs in output position 16.

I/p: Array A[1...n], require two other Array B[1...n] hold the sorted O/p and the Array C[1...K], where K is the highest no. in Array A.

Algo: CountingSort(A, B, K)

```
1- for i ← 1 to K.                    ——— O(K)
2-     C[i] ← 0
3- for J ← 1 to length[A]             ——— O(n)
4-     C[A[J]] ← C[A[J]] + 1
5- for i ← 2 to K.                    ——— O(K)
6-     C[i] ← C[i] + C[i-1]
7- for J ← length[A] down to 1        ——— O(n)
8-     B[C[A[J]]] ← A[J]
9-     C[A[J]] ← C[A[J]] - 1
```



#TC: Total cost = O(K+n) = O(n+K) = O(n)

"max number"

#SC: space = O(n+K) = O(n) ← not inplace sort

Radix Sort:- Radix sort is a sorting Algo that is useful when there is a constant 'd' such that all the keys are 'd' digit number to execute radix sort, for P = 1 to and 'd' sort the number with respect to the pth digit from the right using a linear time stable sort (Counting sort)

Algo: RadixSort(A, d)

```
1- for i ← 1 to d.
2-     do use a stable sort to sort array A on digit i
```

Ex:

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

⇒ ⇒ ⇒ Sorted.

#TC:- $O[d(n+K)] = O(n)$ (d passes, n+K total no.) = $O(dn)$

Let $d = \log_r n$

Then #TC:- $O(n \log_r n)$

#SC: O(n)

n = 4 (00, 01, 10, 11) K = max No

$d = \log_2^4 = \log_2^3 = 2$

No of Digit required to represent n No = $\log_2 n$

For Binary = $\log_2 n$

For Decimal = $\log_{10} n$

No of digit for Radix r is $\Rightarrow d = \log_r n = \lceil \log_r n \rceil$

Ex: If n = 100

Then K = 99

No of Digit = $\lceil \log_{10}^{99} \rceil$

= 2

for n = 10, K = 9

$d = \lceil \log_{10} K \rceil \lceil \log_{10} 9 \rceil$

= 1

Sort n integers in the range "0 – n^2" and find the time complexity using

i) Counting sort → $O(n+K) = O(n+n^2) = O(n^2)$

ii) Radix sort : → $O(d(n+K)) = O(\log_{10} n\,(n+9)) = O(n \log_{10} n)$ ✓

NOTE:- Radix sort is suitable for very large Numbers.

Example: If we used Radix sort to sort n integers in the range $(n^{K/12}, n^K)$ for some $K > 0$ which is independent of n, the time taken would be

a) $\theta(n)$ b) $\theta(Kn)$ c) $O(n \log n)$ d) $\theta(n^2)$

$$\text{Radix sort} = O(d(n+K)) = O(\log_{10} n^K (n+9)) = O(K \log n \times n) = O(n \log n)$$

Example: How to sort n numbers in the range $[0, n^2-1]$ in $O(n)$ time what should be the radix of No r=?

Radix sort = $\theta(n \cdot d)$ for $O(n)$ d should be 1

~~$d = \log_r n^2 = 2 \log_n n = 2$~~

Already given Total no = n

$d = \log_r n$, we have to make d as constant means d=1

Then for Radix should be n

$d = \log_n n = 1 \Rightarrow \boxed{r = n}$ Ans Radix = n.

Sorting Algo:

Sorting	Time: BC	WC	AC	Inplace	STABLE	space: BC	WC	AC
B.S.	$O(n)$	$O(n^2)$	$O(n^2)$	✓	✓	$O(1)$	$O(1)$	$O(1)$
S.S	$O(n^2)$	$O(n^2)$	$O(n^2)$	✓	X	$O(1)$	$O(1)$	$O(1)$
I.S	$O(n)$	$O(n^2)$	$O(n^2)$	✓	✓	$O(1)$	$O(1)$	$O(1)$
m.S	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	X	✓	$O(n)$	$O(n)$	$O(n)$
Q.S	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	X	X	$O(\log n)$	$O(n)$	$O(\log n)$
H.S.	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	X	$O(1)$	$O(1)$	$O(1)$

↳ Worstcase lower Bound = $O(n \log n)$

↳ BestCase Upper Bound = $O(n^2)$

Time ∝ 1/space