Merge Sort:

- it is an Divide & Conquer Algorithm
  (divide the array in two parts, sorts these two parts, then merge this two parts)
- Stable
- O(nlogn) ← TC | O(n) ← SC (not in-place)
- Block Merge Sort:
  D is a sorting algo combining atleast two merge operations with an insertion sort to arrive at O(nlogn) in-place stable sorting algo.
- For Array Quick Sort better than MS
- ✓ For Linked list MS better uses O(1) Aux space
- Well suited for external sorting (bring parts of array in RAM then sort)

(II) Merge Without Extra Space O(1) → SC

I/p → a1 [1 3 5 7] | o/p → a1 [0 1 2 3]
a2 [0 2 6 8 9] | a2 [5 6 7 8 9]

- Given two sorted array, sort in-place.

a) Take a new temp[] array copy element of a1 & a2 in temp[] then sort temp[]
TC → O((n+m)log(m+n)) SC → O(n+m)

b) TC → O(nm)

a1[] = 1 4 7 8 10 (i)
a2[] = 2 3 9 (j)

Insertion sort concept or simple for loop swap ↑

If(a1[i] > a2[j]) swap;
& ~~sort~~ put a2[j] element in correct position in a2 array

here 4>2

swap ⇒ 1 2 7 8 10 / 4 3 9 ⇒ 4 at correct position in arr 2 ⇒ 1 2 7 8 10 / 3 4 9

Doubt gap का cell why ?

classmate Date Page

**

Q) Gap Method (Shell ~~Sort~~ sort) TC- $O((n+m)\log(n+m))$ ← while loop

```
void merge(long long a1[], long long a2[], int n, int m)
{
    int i, j, gap = ceil((float)(n+m)/(float)2);
    while(gap>0)
    {
        for(i=0; i+gap<n; i++)
        {
            if(a1[i] > a1[i+gap])
                swap(a1[i], a1[i+gap]);
        }
        for(j=gap>n ? gap-n : 0; i<n && j<m;
                                    i++, j++)
        {
            if(a1[i] > a2[j])
                swap(a1[i], a2[j]);
        }
        for(j=0; j+gap<m; j++)
        {
            if(a2[j] > a2[j+gap])
                swap(a2[j], a2[j+gap]);
        }
        if(gap==1)
            break;
        gap = ceil((float)gap/(float)2);
    }
}
```

एक situation जो while loop में एक gap का different रहेगा।

हम चाहते हैं कि gap=1 पर एक ही बार चले

when gap=1 then ceil(1/2) is always 1 causing TLE.

eg a1 [1 3 5 7]
a2 [0 2 6 8 9]

* 1st for loop : comparing element in first array a1
2nd ——— : comparing element in both array a1, a2 with help of two pointers.
3rd ——— : comparing element in second array itself a2.

(III) Merge Sort :

```
void merge(int a[], int l, int m, int r)
{   int n1 = m-l+1, n2 = r-m;
    int left[n1], right[n2];
    for(int i=0; i<n1; i++)
        left[i] = a[l+i];
    for(int j=0; j<n2; j++)
        right[j] = a[m+1+j];
    int i=0, j=0, k=l;      // k=l
    while (i<n1 && j<n2)
    {
        if( left[i] <= right[j]){
            a[k++] = left[i++]; }
        else
            a[k++] = right[j++];
    }
    while(i<n1) { a[k++] = left[i++]; }
    while(j<n2) { a[k++] = right[j++]; }
}
void mergeSort( int a[], int l, int r)      // l=0, r=n-1;
{
    if(r>l)
    {   int m = (l+r)/2;
        mergeSort(a, l, m);
        mergeSort(a, m+1, r);
        merge(a, l, m, r);
    }
}
```

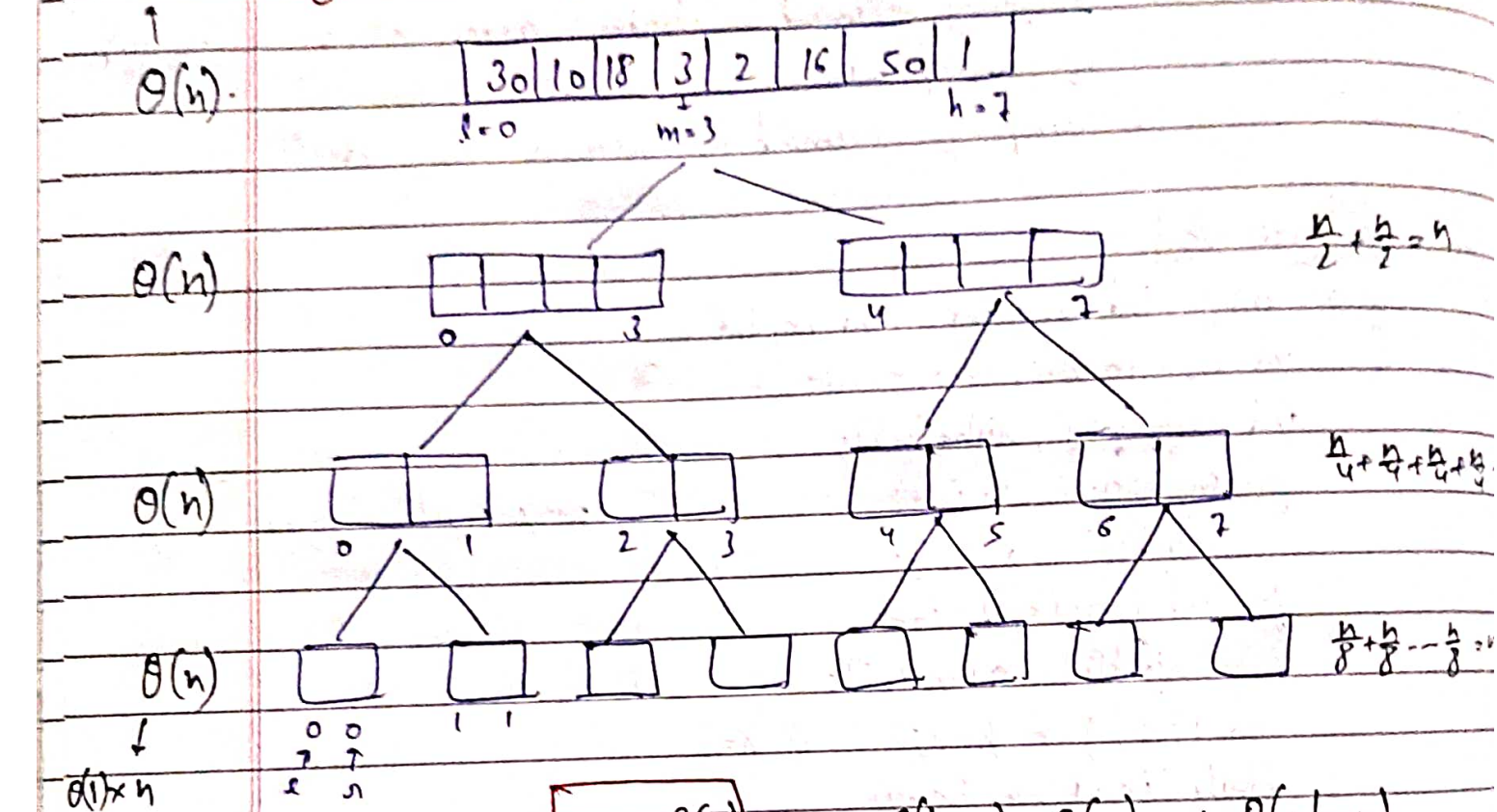
making left & right array

→ merge Algo
→ merge will happen only when a[] has atleast 2 element

Work done at every level is $O(n)$

$O(\frac{n}{2}) + O(\frac{n}{2})$

Merge Sort Analysis :



if (l < r) false
∴ $O(1)$ work

$x * O(n) \Rightarrow O(\log n) * O(n) \Rightarrow O(n \log n)$

(x = no. of levels)

here $n = 8$ $(\log_2 8 + 1) = \cancel{3 \text{ levels}} = 3 + 1 = 4$ levels

Space Complexity $\Rightarrow O(n)$

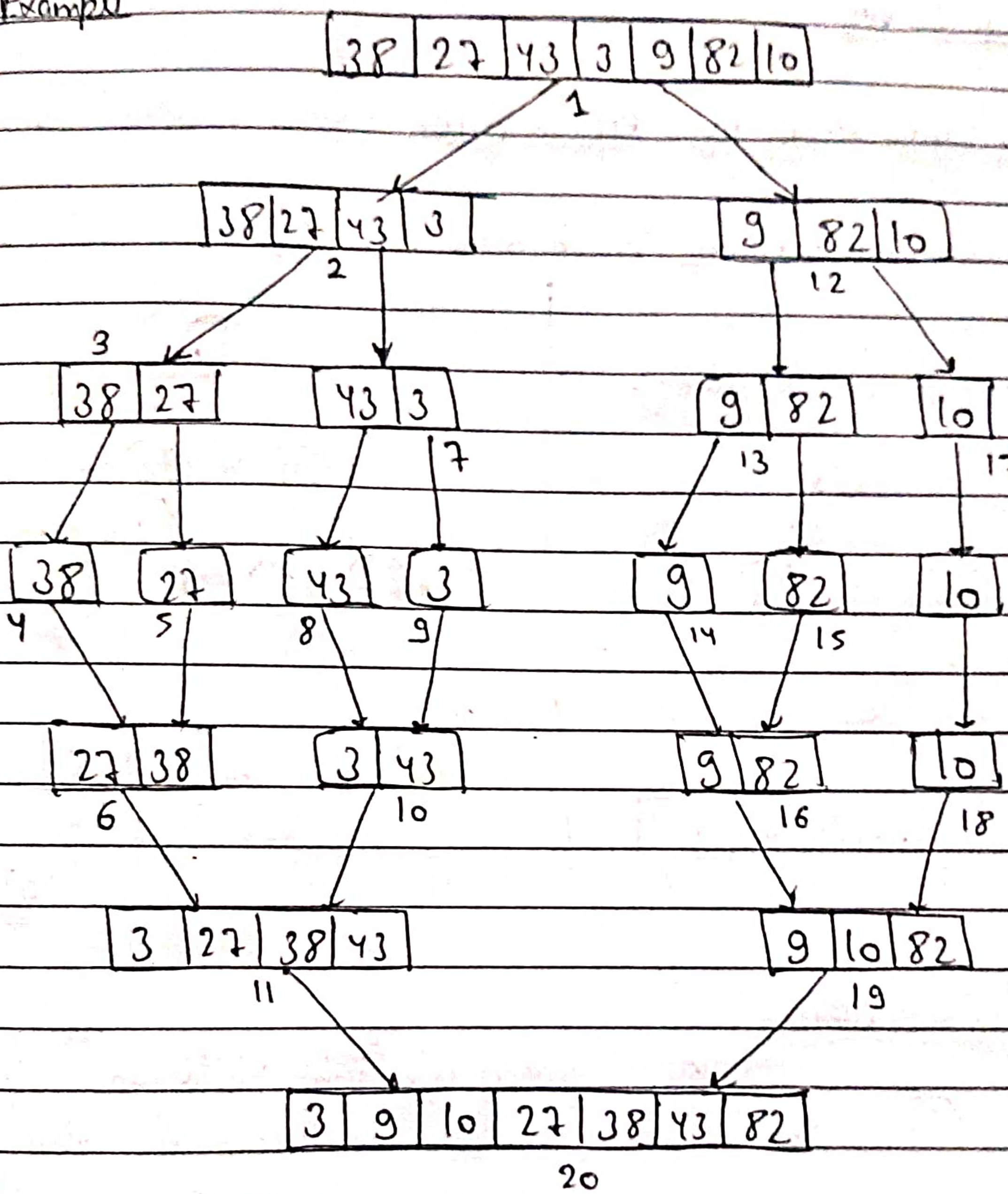
Merge() takes $O(n)$ time

→ when n is even no. of levels $(\log_2 n + 1)$

→ when n is odd ——— $\lceil \log_2 n \rceil + 1$

(ceil value)

Example



Merge procedure

Example

2	4	5	7

1	2	3	6

- make two array
- make size n+1
- make last element ∞

L

2	4	5	7	∞

i

R

1	2	3	6	∞

j

if i<=j
then A[k]=i
& i++
else
A[k]=j
j++

ये array A वही है जो Given array होता है

← A

1	2	2	3	4	5	6	7

final array

(IV) Count Inversions :

I/p: n=5, a[] = {2, 4, 1, 3, 5}	O/p: 3	(2,1), (4,1), (4,3) ↓ possible inversions

- Two element a[i], a[j] form an inversion if a[i] > a[j] & i < j

(I) Brute force : $O(n^2)$ $S(-O(1)$

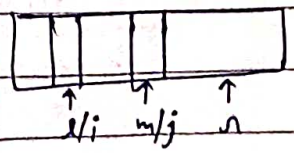
(II) Merge Sort : $O(n \log n)$ $SC-O(n)$

Use long long

```
int inversionCount(int a[], int n)
{
    int b[n];
    return mergeSort(a, b, 0, n-1);
}

int mergeSort(int a[], int b[], int l, int r)
{
    int m = (l+r)/2, count=0;
    if(l<r)
    {
        count += mergeSort(a, b, l, m);
        count += mergeSort(a, b, m+1, r);
        count += merge(a, b, l, m, r);
    }
    return count;
}

int merge(int a[], int b[], int l, int m, int r);
{
    int count=0, i=l, j=m+1, k=l;
    while(i<=m && j<=r)
    {
        if(a[i] <= a[j])
            b[k++] = a[i++];
        else
        {
            b[k++] = a[j++];
            count += (m+1-i);
        }
    }
```



```
while(i<=m)  b[k++] = a[i++];
while (j<=n)  b[k++] = a[j++];
for(i=l; i<=n; i++)
    a[i] = b[i];
return count;
}
```

Quick Sort: Partition a given Array:

(I) Naive Approach

I/P: {3, 8, 6, 12, 10, 7} if pivot = 5

O/P: {3, 6, 7, 8, 12, 10}

or

{6, 3, 7, 12, 8, 10}

or

so on

7 से left में सारे छोटे

7 से right ——— बड़े

```
void partition(int a, int l, int h, int p)
{
    int temp[h-l+1], index = 0
    for(int i=l; i<=h; i++)
        if(a[i] <= a[p])
        { temp[index] = a[i]; index++; }

    for(int i=l; i<=h; i++)
        if(a[i] > a[p])
        { temp[index] = a[i]; index++; }

    for(int i=l; i<=h; i++)
        a[i] = temp[i-l];
}
```

TC → O(n)

SC → O(n)

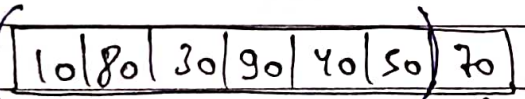
last element is pivot

(II) Lomuto Partition: TC → O(n) SC → O(1)

```
int partition(int a[], int l, int h)
{   int p=a[h];
    int i=l;
    for(int j=l; j<=h-1; j++)
    {
        if(a[j]<p)
        {
            swap(a[j],a[i]);
            i++;
        }
    }
    swap(a[i],a[h]); // swap the pivot(a[h]) with first greater element than pivot.
    return i;
}
```

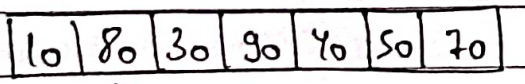
* in LP we were picking the pivot & fixing the pivot at its correct position

eg:

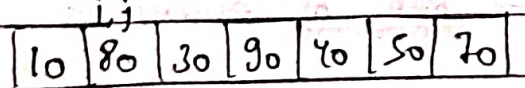


i=0, j=0, h=1, h

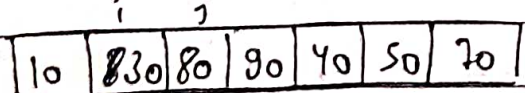
for loop



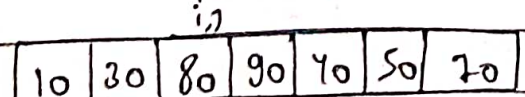
10<70 swap & i++ i=1, j=1



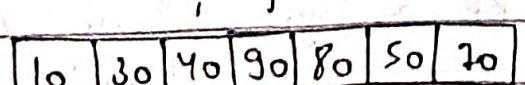
i=1, j=2



30<70 swap i++, i=2, j=2



j=3, i=2

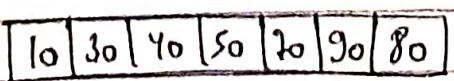


j=4 40<70 swap i++ i=3

50<70 swap i++ i=4, j=5

in next iteration j=6 >= h=1

at last swap(a[i] & a[h])



return i=4

→ When pivot is not the last element, then simply swap() the pivot with last element

First element is pivot

(III) Hoare's Partition TC → O(n) SC → O(1)

- HP takes less no. of comparision than Lomuto Partition.
- It doesn't fix the pivot at its correct position
- Corner Case: pivot is smallest {5, 10, 12, 14} (i, j)
  pivot = largest ——→ {12, 10, 5, 9} → {9, 10, 5, 12}
  all elements are same — only one swap

```
int partition(int a[], int l, int h)
{
    int pivot = a[l];
    int i=l-1, j=h+1;
    while(1)
    {
        do{ i++; } while(a[i] < pivot);
        do{ j--; } while(a[j] > pivot);

        if(i>=j) return j;
        swap(a[i], a[j]);
    }
}
```

* swap करने के बाद i++ और j-- क्यों नहीं because do while loop.

Corner Case: pivot is largest

{12, 10, 5, 9} → {12, 10, 5, 9} —swap→ {9, 10, 5, 12} → {9, 10, 5, 12}
 i j i j i j j i

pivot = 5 * {1, 3, 2, 4}, {8, 7, 5, 10} in HP on left (<= pivot) elements are in right (>= pivot)
 j i <=j >=j

pivot = 5 * {5, 3, 2, 5, 8, 7, 9, 10}
 j i

→ Hoare & Lomuto both Unstable

Unstable ✓ Corner Case: all elements are same

{5, 5, 5, 5} i=-1, j=4

{5, 5, 5, 5} swap ⇒ {5, 5, 5, 5} swap ⇒ {5, 5, 5, 5} return j

eg: | 5 | 3 | 8 | 4 | 2 | 7 | 1 | 10 | i=-1, j=8

| 5 | 3 | 8 | 4 | 2 | 7 | 1 | 10 | swap a[i] & a[j]

| 1 | 3 | 8 | 4 | 2 | 7 | 5 | 10 | "

| 1 | 3 | 2 | 4 | 8 | 7 | 5 | 10 | return j

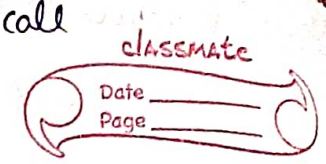
Quick Sort

- Divide and Conquer Algo (divide array in 2 part then sort these 2 part independently)
- Worst Case $O(n^2)$
- Consider faster than Merge Sort b/z it is in-place, cache friendly, average case $O(n \log n)$, tail recursive
- If you don't need stability Quick Sort is fastest Algo

(I) QS using LP:

```
void QuickSort(int a[], int l, int h)
{
    if(l<h)
    {
        int b= partition(a, l, h);
        QuickSort(a, l, b-1);
        QuickSort(a, b+1, h);
    }
}
```

Tail Recursion: No extra work after child function call



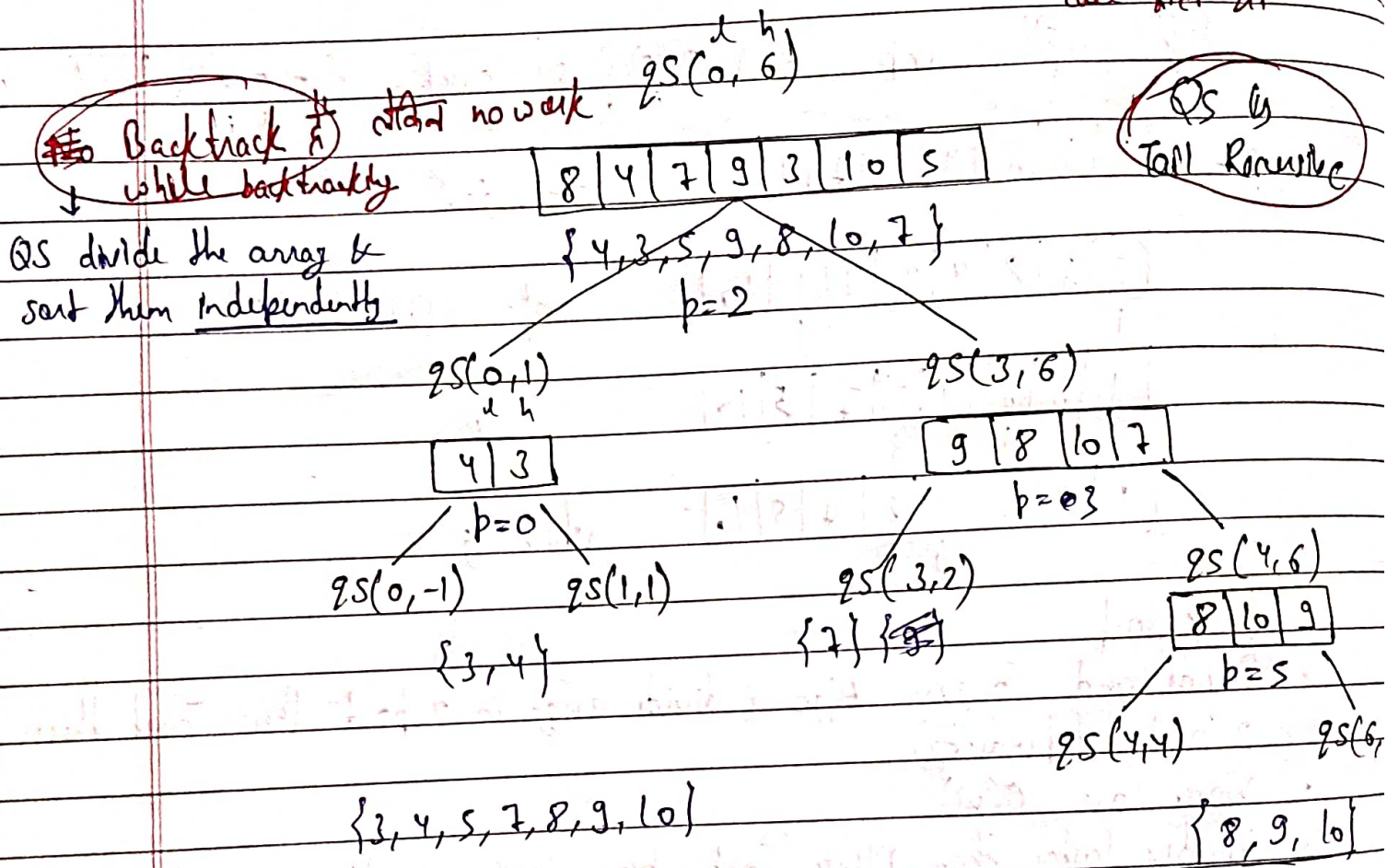
eg: {8, 4, 7, 9, 3, 10, 5}
0 1 2 3 4 5 6

Merge Sort
Backtrack करते # merge() function call होता था

Backtrack # no work while backtracking

QS divide the array & sort them independently

QS is Tail Recursive



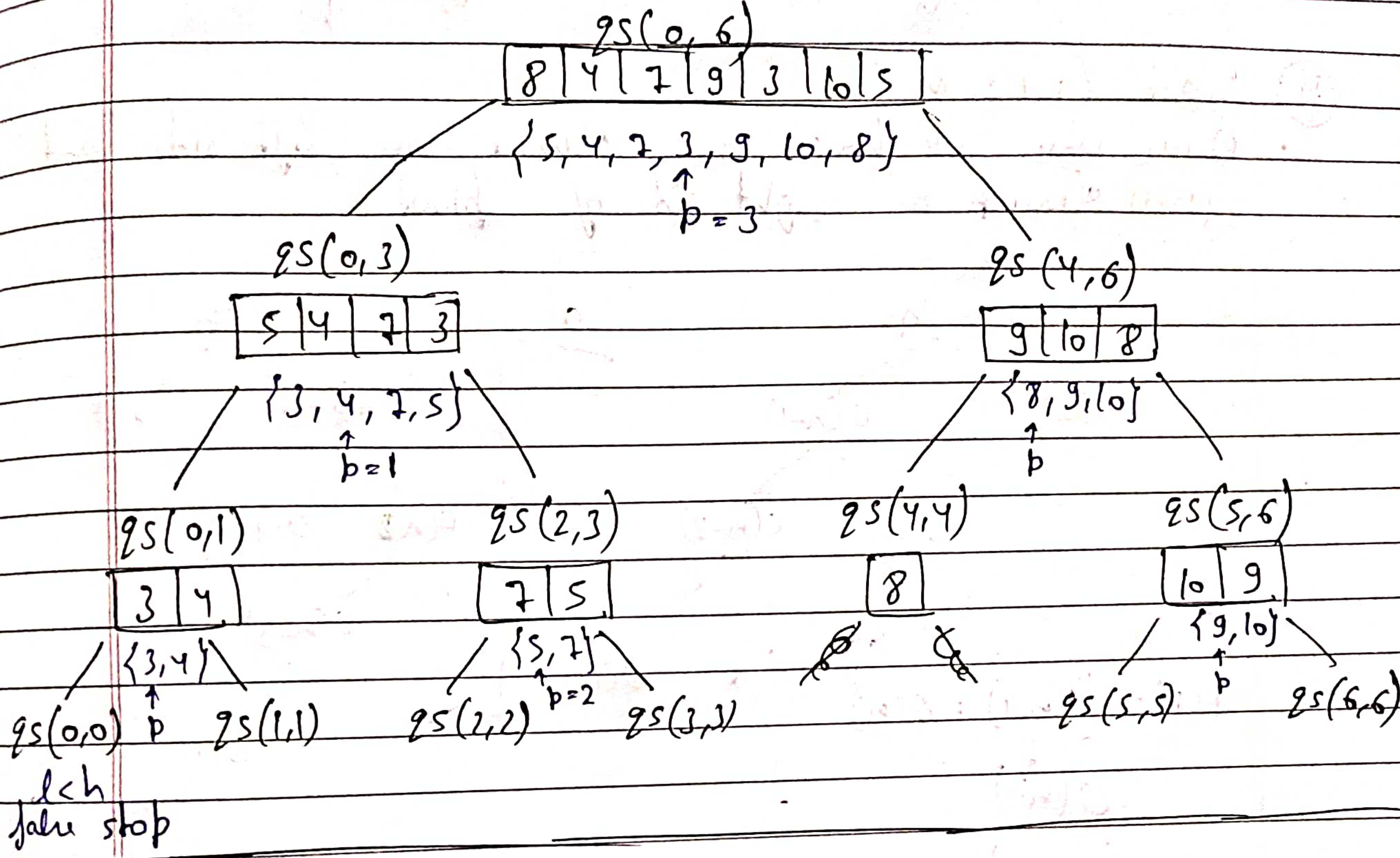
(II) QS using HP:

```
void QuickSort(int a[], int l, int h)
{
    if(l<h)
    {
        int p= partition(a, l, h);
        QuickSort(a, l, p);
        QuickSort(a, p+1, h);
    }
}
```

*

→ See properly the returned position of p

eg: {8, 4, 7, 9, 3, 10, 5} (indices 0 to 6; l at index 0, h at index 6)

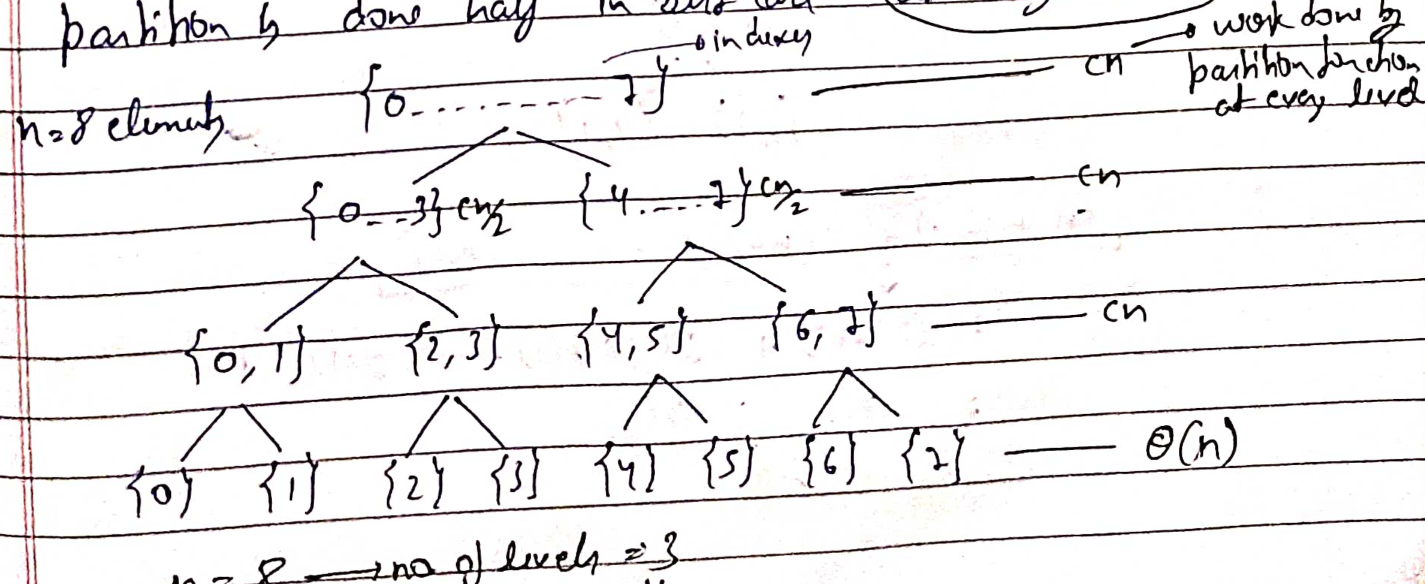


→ **Analysis of QuickSort:**

- Analysing using HP (for both it is same)

(I) Best case for QS

→ partition is done half in best-case at every level



n = 8 → no of levels = 3

n = 16 → ← = 4

When Array is sorted both HP & LP will give do worst case complexity

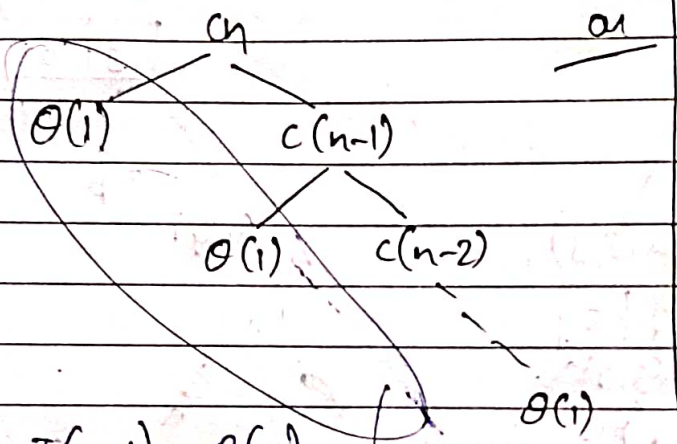
$cn + cn + cn + \cdots + \theta(n)$

$cn \times \log_2 n + \theta(n) = \theta(n \log_2 n)$

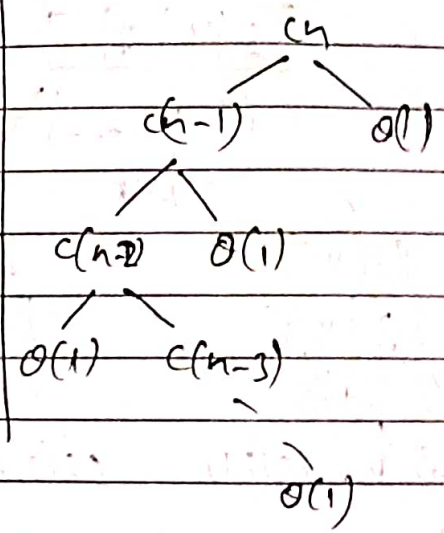
$T(n) = 2T(\frac{n}{2}) + \theta(n)$

(II) Worst case:

- Occur when we have one element on left side and (n-1) element on right side of pivot.



or



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

$\downarrow$

$\theta(n^2)$

eg: {0, 4}

{0,0} {1,4}

{1,1} {2,4}

{2,2} {3,4}

{3,3} {4,4}

no. of nodes = 5 → no. of levels = 5

$cn + c(n-1) + c(n-2) + \cdots \theta(1)$

$\downarrow$

$= \theta(n^2) + \theta(1) + \theta(1) \cdots \theta(1)$

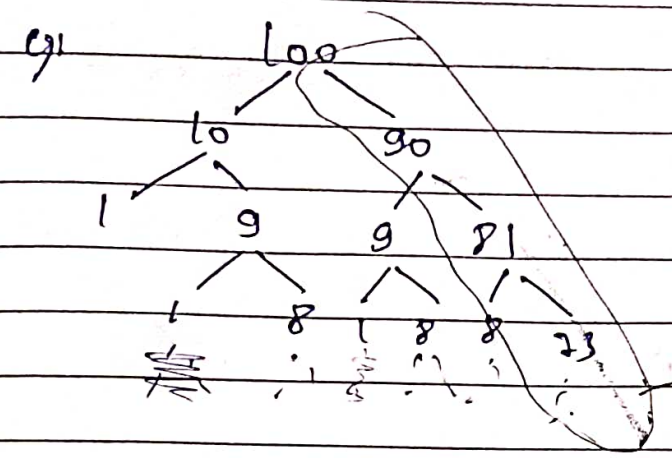
$= \theta(n^2) + \theta(n)$

$= \theta(n^2)$

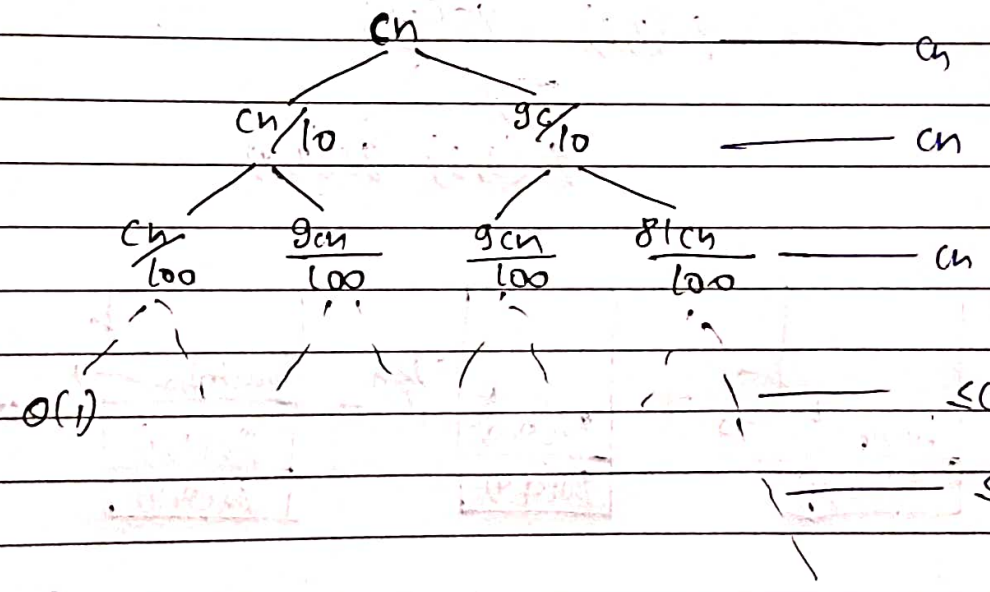
Random pivot selection gives $O(n \log n)$

(III) Average Case :

- Here we calculate AC by dividing the array into $\left(\frac{n}{10}\right)$ on one ~~left~~ side & $\left(\frac{9n}{10}\right)$ element on ~~right~~ other side



- left side of subtree will do the smallest work & right side will do largest work



$$O\left(cn \times \frac{\log n}{\log_{10/9}}\right) = O(n \log n)$$

↑ upper bound

cn → work done at every levels

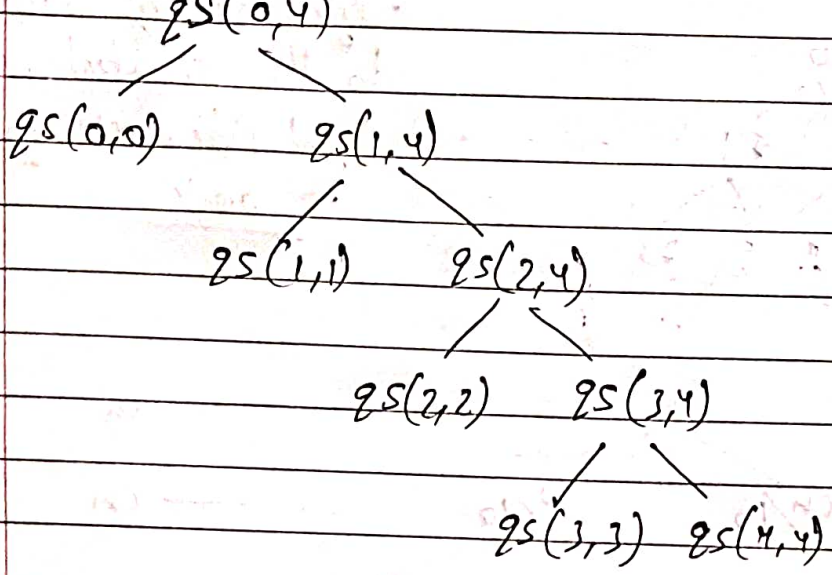
$\log_{10/9}$ → no. of levels

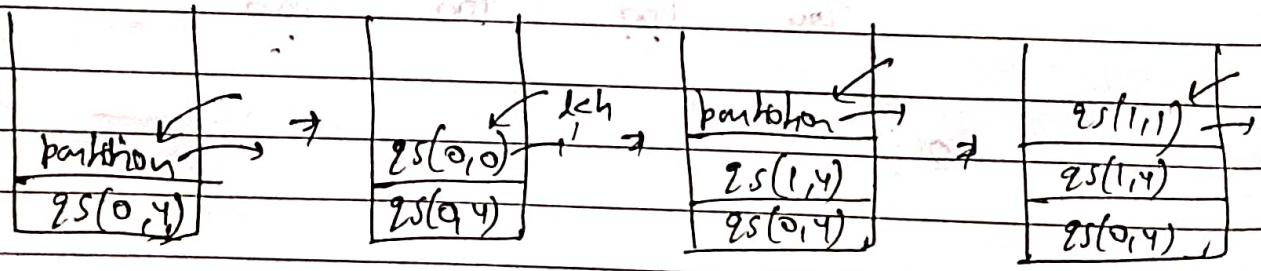
- when divide cn by $cn/2$ we have $\log_2 n$ levels
- ——— cn ——— $cn/\frac{10}{9}$ we have $\log_{10/9}$ levels
- at last levels there will be $< cn$ work but here in calculation we do $cn \times \log n$

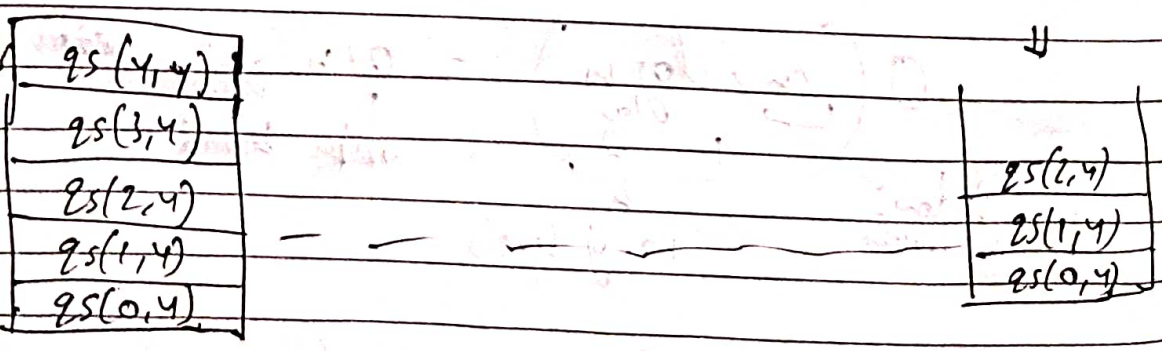
→ Space Analysis of QS

- Require extra space for recursion call stack.
- Partition takes O(1) space.
- QS is

Worst Case







at max no. of function call in the function call stack

SC → O(4)

Adversary: when the algo fail for some pattern or give worst case complexity

classmate Date Page

Best Case (for diagram see TC tree)

- occur when divide the array in two half
  $n = 8 \Rightarrow$ ~~no of levels~~ no. of function call at max in function call stack is $\log_2 8 + 1 = 4$

$SC \rightarrow O(\log n)$

** When Arr is sorted choosing last or first element as Pivot will give Worst case Complexity

→ To avoid WCC we use rand() function to generate random pivot.

→ And swap with either first or last element. according to LP or HP.

(V) Kth Smallest element

I/p → a[] → 7 10 4 3 20 15 k=3
o/p → 7

a) Naive → sort

b)
```
int KthSmallest(int a[], int l, int r, int k)
{
    while(l<=r)
    {   int p1 = rand() % (r-l+1);          ] rand() implement
        swap( a[r], a[l+p1]);               ]
        int p = partition(a, l, r);
        if( p==k-1)                          p is left of
            return a[p];                     all smaller element
        else if (p> (k-1))                   & pth is the required
            r=p-1;                           element.
        else
            l=p+1;
    }
}
```

Lomuto partition

```
int partition( int a[], int l, int r)
{
    int p= a[r], i=l;
    while for(int j=0; j<= r-1; j++)
    {
        if( a[j] < p)
        {
            swap( a[j], a[i]);
            i++;
        }
    }
    swap( a[i], a[r]);
    return i;
}
```

(VI) Union of Two Sorted Arrays :

I/p n=5 a[] = {1, 2, 3, 4, 5} o/p : 1 2 3 4 5
m=3 a2[] = {1, 2, 3}

```
vector<int> f(int a[], int b[], int n, int m)
{
    vector<int> v;
    int l1=0, l2=0;
    while(l1<n && l2<m)
    {
        if(a[l1]==a[l1+1]) { l1++; continue; }
        if(b[l2]==b[l2+1]) { l2++; continue; }
        if(a[l1]==b[l2])
        { v.push_back(a[l1]); l1++; l2++; continue; }

        if(a[l1]<b[l2])
        { v.push_back(a[l1]); l1++; }
        else if(a[l1]>b[l2])
        { v.push_back(b[l2]); l2++; }
    }
    while(l1<n)
    { if(v.size()>0 && v[v.size()-1]!=a[l1])
          v.push_back(a[l1]);
    } l1++;

    while(l2<m)
    { if(v.size()>0 && v[v.size()-1]!=b[l2])
          v.push_back(b[l2]);
      l2++;
    }
    return v;
}
```

avoiding duplicates