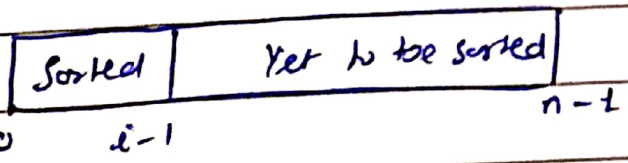
⇒ Insertion Sort

1. $O(n^2)$ worst case
2. In-Place and Stable
3. Used in practice for small arrays (and in TimSort and IntroSort).
4. $O(n)$ is Best case.

```
void isort (int arr[], int n)
{   for (int i=1 ; i<n; i++)
    {   int key = arr[i];
        int j = i-1;
        while (j>=0 && arr[j] > key)
        {   arr[j+1] = arr[j]
            j--;
        }
        arr[j+1] = key;
    }
}
```



↑ Start from here

If sorted part become whole array, then array is sorted

Scanned with CamScanner

{50, 20, 40, 60, 10, 30}

i = 1, Key = arr[i] = 20

j = 0,

↓

[50, 50, 40, 60, 10, 30]

j = -1

↳ out of loop

arr[j+1] = Key

↳ arr[0] = 20

↓

[20, 50, 40, 60, 10, 30]

i = 2, key = arr[2] = 40

j = 1

↓

{20, 50, 50, 60, 10, 30}

↑

j = 0 ↳ stopped.

arr[1] = key = 40

↓

{20, 40, 50, 60, 10, 30}

i = 3, Key = arr[i] = 60

j = 2, ◯ overwrite - itself

~~{20, 40, 50, 60, 10, 30}~~

↑ stopped

arr[j+1] = arr[3] = Key = 60

i = 4, Key = arr[i] = 10

j = 3

{20, 20, 40, 50, 60, 30}

↑

stopped

j ↳ become - ive.

arr[j+1] = Key => arr[0] = 10

{10, 20, 40, 50, 60, 30}

i = 5, Key = arr[i] = 30

j = 4 {10, 20, 40, 40, 50, 60}

↑ stopped

arr[j+1] = 30 => {10, 20, 30, 40, 50, 60}

Scanned with CamScanner

Best Case:

↺ ↺ ↺ ↳ when array is sorted

[10, 20, 30, 40] i.e. O(n)

bcz, we never went into inner loop

Worst Case:

when array is reverse sorted

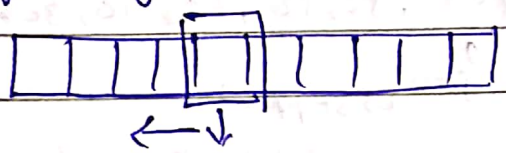
{ 50, 40, 30, 20, 10 } $O(n^2)$

Auxillary Space : O(1)

⇒ Increasing Efficiency.

1st Approach

Insertion Sort



For comparison :→ Minimum : 1 comparision

Maximum : n comparision

Binary → Now if we replace the ~~Se~~ comparision part with binary search we get

Mini :- 1 Comparision

Max :- $\log n$ comparision

→ It create new algorithm called binary insertion sort.

New for worst random case, ~~on~~ time complexity will remain Same, $O(n^2)$ bcz of swapping loop.

But in worst case where array is in ~~descending~~ order, time complexity will be ~~of~~ long $O(n \log n)$.

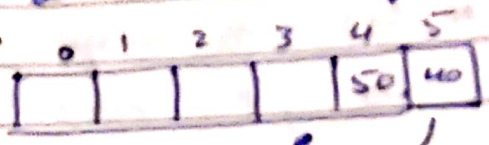
Scanned with CamScanner

2nd Approach

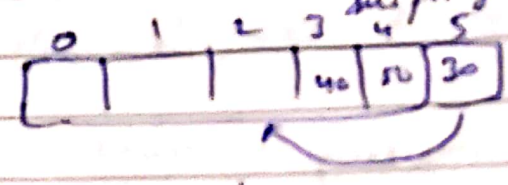
To get rid of swapping loop → i.e. shifting of element to insert an element, we can create an auxiliary array of size a new_arr[n+1]. and we store inserting at the end of this array.

eg. 50 | 40 | 30 | 20 | 10

new_arr[n+1] = arr[0]



shifting = 1 (constant time)



Thus will reduce the time complexity in worst case to O(n log n) bz comparison using binary search like (log n).

But in best case, time will still be O(n) And in random case, time will tends to O(n²) => upper bound

O(n²)

Scanned with CamScanner

=> Bubble Sort

Logic :-

We start from index = 0, and compare it with all other elements till n-1. If a[i] < a[0] we swap.
Same goes for index = 1, 2, ---, n-1.

It requires two for loops to implement.

Time complexity : $O(n^2)$ ⇒ worst

In-space Aux-space O(1)

Modification :-

It might happen that array get sorted before first loop reach its end.
So to exit, we can use flag.

Set flag = false → when we enter inside first for loop.
Now if we go inside second for loop we set flag = true → i.e., if swapping occurs. set

and after coming out of it, check

```
if (!flag)
    return;
```

In this case, best time complexity will be O(n).

Scanned with CamScanner

⇒ Merge Sort

① Divide and Conquer Algorithm
(Divide, Conquer and Merge)
② Stable Algorithm
③ $\Theta(n \log n)$ time and $O(n)$ Aux Space
④ Well suited for linked lists. Works in $O(1)$ Aux Space.
⑤ Used in External Sorting.
⑥ In general for arrays, Quicksort outperforms merge sort.

⇒ Merge Two Sorted Arrays

I/P: a[] = {10, 15, 20, 40}
b[] = {5, 6, 6, 10, 15}
O/P: 5 6 6 10 10 15 15 20 40
I/P: a[] = {1, 1, 2}
b[] = {3}
O/P: 1 1 2 3

Method 1 :-
(Naive)

```
void merge(int a[], int b[], int m, int n)
{ int c[m+n];
O(m) for(i=0; i<m; i++) { c[i] = a[i]; }
O(n) for(j=0; j<n; j++) { c[j+m] = b[j]; }
O((m+n)*log(m+n)) sort(c, c+m+n);   // (STL)
O(m+n) for(int i=0; i<m+n; i++)
        { cout << c[i] << " "; }
}
```

Time Complexity
⇒ $O((m+n) * \log(m+n))$

Scanned with CamScanner

Better Approach :-

```
void merge (int a[], int b[], int m, int n)
{   int i=0, j=0;
    while (i<m && j<n)
    {   if (a[i] < b[j]) { cout << a[i++] << " "; }
        else { cout << b[j++] << " "; }
    }
    while (i<m) { cout << a[i++] << " "; }
    while (j<n) { cout << b[j++] << " "; }
}
```

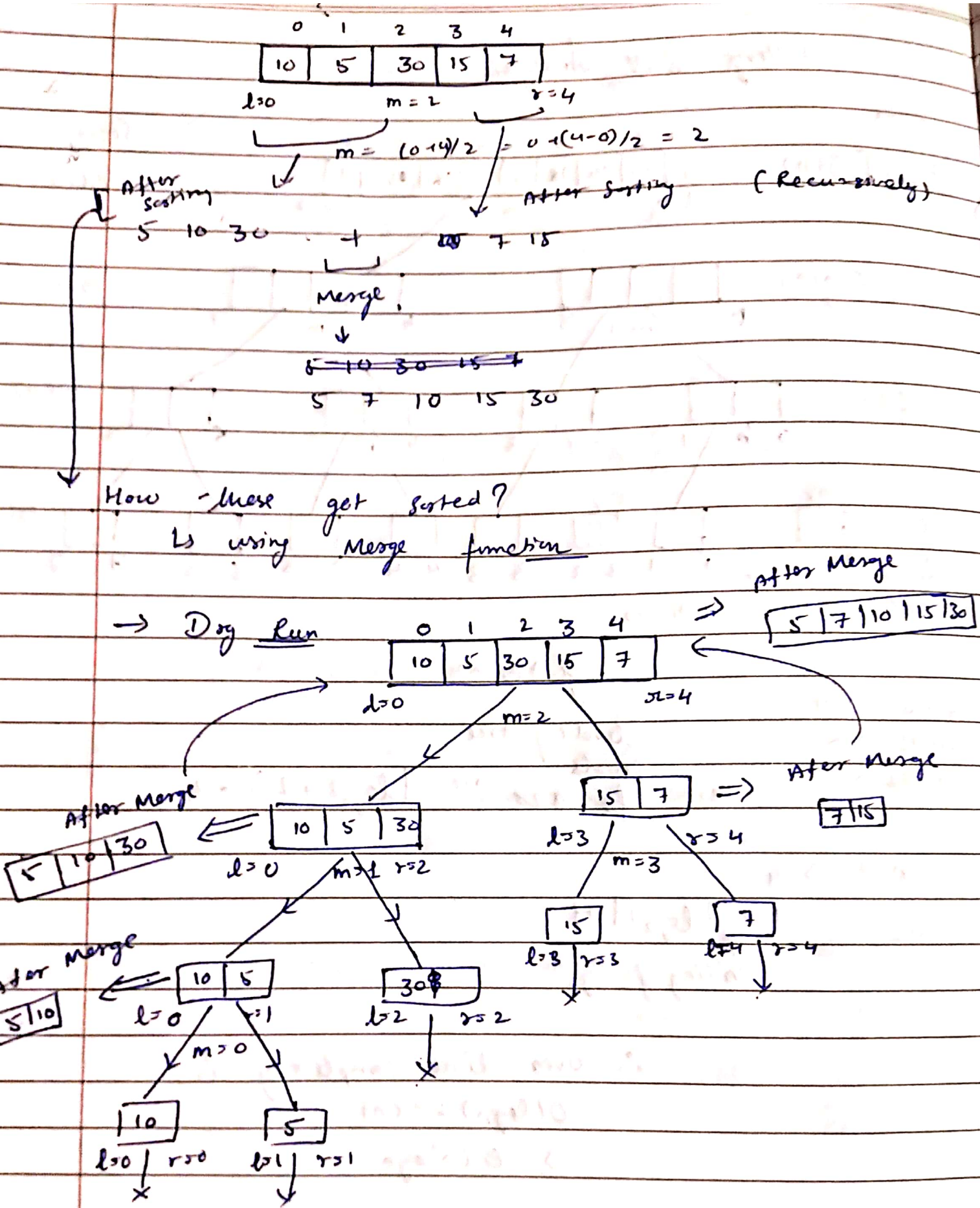
for those cases in which array i==m (=m) w/P/

Time Complexity : O(m+n)

⇒ Merge Sort Algorithm

```
void mergeSort (int arr[], int l, int r)
{   if (r>l)        // Atleast 2 elements.
    {   int m = l+(r-l)/2;      // (l+r)/2
        mergeSort (arr, l, m);
        mergeSort (arr, m+1, r);
        ~~mergeSort (arr~~
        merge (arr, l, m, r);
    }
}
```

we use this instead of (l+r)/2 is avoid overflow in array when l and r are high.

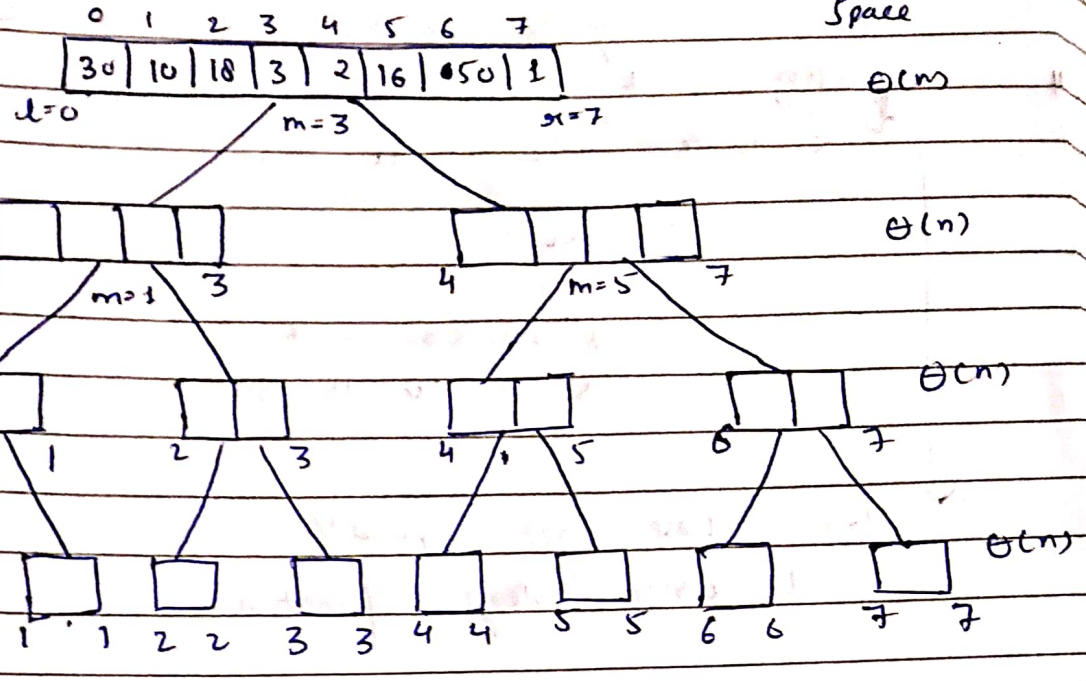


Scanned with CamScanner

⇒ Merge Sort Analysis

Time

for merging [$\Theta(n)$



$\Theta(1) + \Theta(1) + \Theta(1) + \dots$ n times

$n * \Theta(n)$

$n = 8 \Rightarrow 4$

$n = 16 \Rightarrow 5$

$n = 32 \Rightarrow 6$

here levels are $(\log_2 8 + 1) = 4$

$n = 7 \Rightarrow 4$

↳ $\lceil \log_2 7 \rceil + 1$

ceiling of $\log_2 n$

∴ over time complexity is,

$\Theta(\log n) * \Theta(n)$

$\Rightarrow \Theta(n \log n)$

Auxillary Space $\Rightarrow \Theta(n) \Rightarrow$ bcz of merge function, bcz it create $\Theta(n)$ array and merge them.

↳ why $\Theta(n)$, why not $\Theta(n \log n) \Rightarrow$ bcz at moment in RAM, we only need $\Theta(n)$ space only, bcz after calling a certain part, it get deallocated and its space is utilized by another call.

Scanned with CamScanner

Date

Quick Sort

Partition function of Quick Sort

⇒ Naive Partition

I/P : arr[] = {3, 8, 6, 12, 10, 7}

pos = 5 // Index of last element

O/P : arr[] = {3, 6, 7, 8, 12, 10}

or

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

or

:

It must partition the array around arr[pos] such that all the numbers smaller than or equal to arr[pos] must come to its left and greater to its right (in any order)

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

Scanned with CamScanner

arr[] = {5, 13, 6, 9, 12, 11, 8}

p = 6
l = 0
h = 5

arr[] = {5, 6, 8, 13, 9, 12, 11}

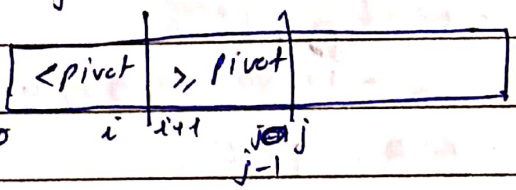
arr[] = {5, 6, 8, 13, 9, 12, 11}

O(n) = Time
Aux Space O(n)

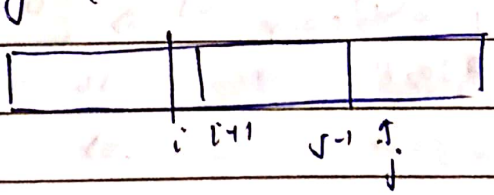
⇒ Lomuto Partition

→ Assume Pivot is always last element

→ We traverse from l to h-1 while ensuring



if (arr[j] < pivot)



↳ then we increase value of i, and swap arr[i], arr[j] => i.e., we increase the

< pivot	Area

Scanned with CamScanner

Date

```
void Partition ( int arr[], int l , int h)
{
    int pivot = arr[h];    // Always last
    int i = l - 1;
    for (int j = l; j <= h-1; j++)
    {   if (arr[j] < pivot)
        {   i++;
            swap (arr[i], arr[j]);
        }
    }
    swap (arr[i+1], arr[h]);
    return (i+1);
}
```

Scanned with CamScanner

Date

⇒ Quick Sort

(1) Divide and Conquer Algorithm
(2) The key part is partitioning (Hoare, Lomuto, Naive) — In-place, out-place
(3) Worst case time complexity is $O(n^2)$
(4) Despite having higher worst case time, it is preffered over other algorithm many times due to following Reasons:
   (a) Tail Recursive
   (b) In-place
   (c) Cache friendly
   (d) Averge Case $\Theta(n \log n)$

⇒ Quick Sort using Lomuto Partition

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

#

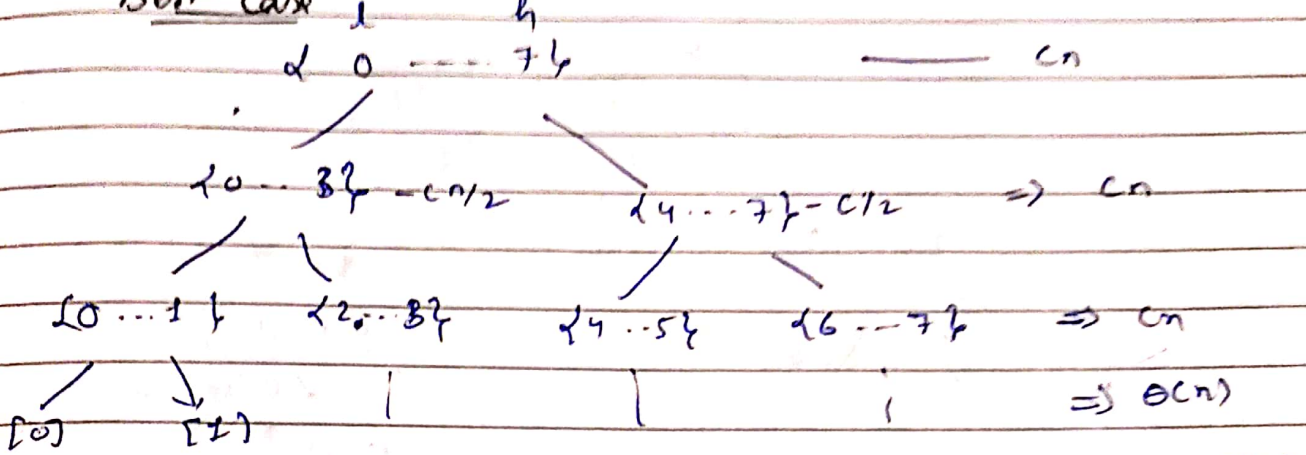
{ 8, 4, 7, 9, 3, 10, 5 }

< 5	5	> 5
0 1 2 3 4 5 6		

Scanned with CamScanner

⇒ Quick Sort analysis (Merge)

Best Case



cn + ... + c0 + cn + cn + θ(n)

8 elements → 3 cn

16 " → 4 cn

cn × log n + θ(n)

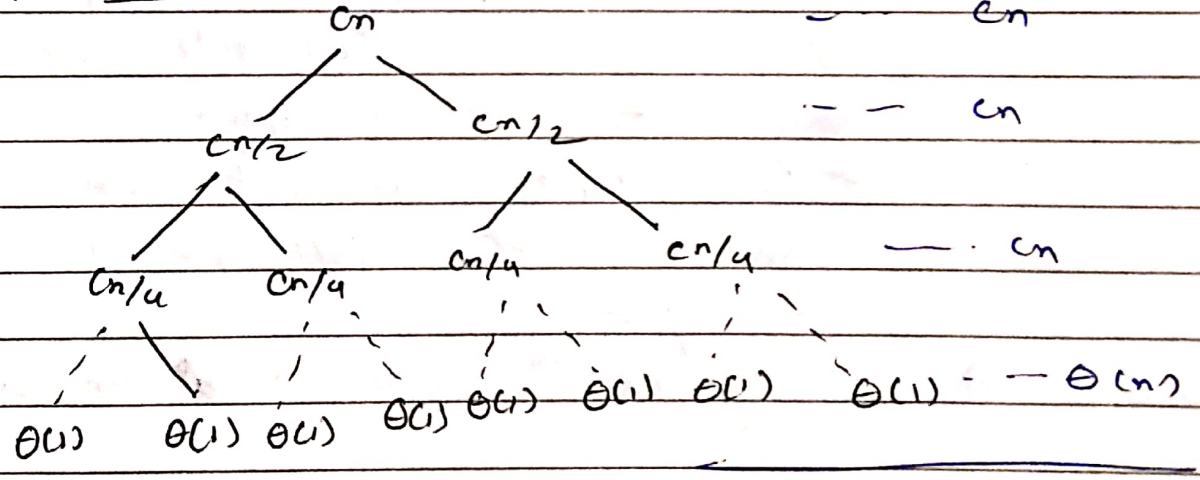
⇒ θ(n log n)

Best case

Best case

T(n) = 2T(n/2) + θ(n)

General Case

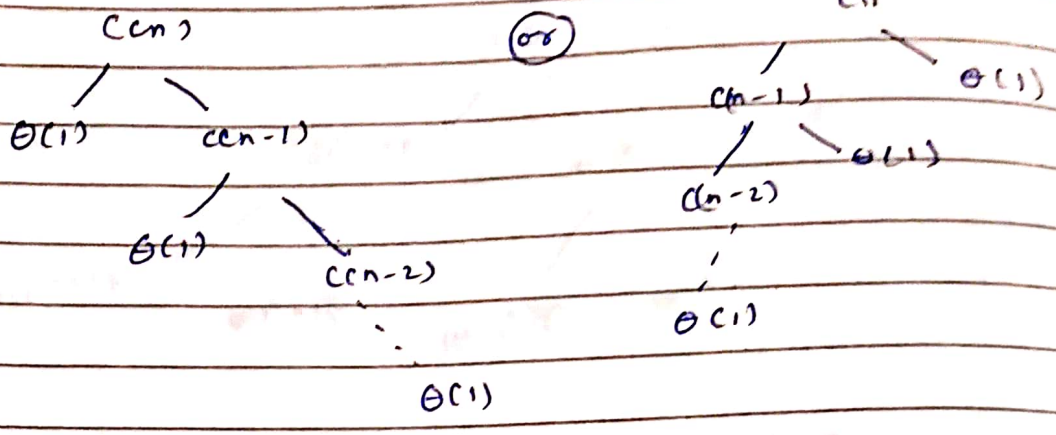


cn + cn + ... cn + θ(n)

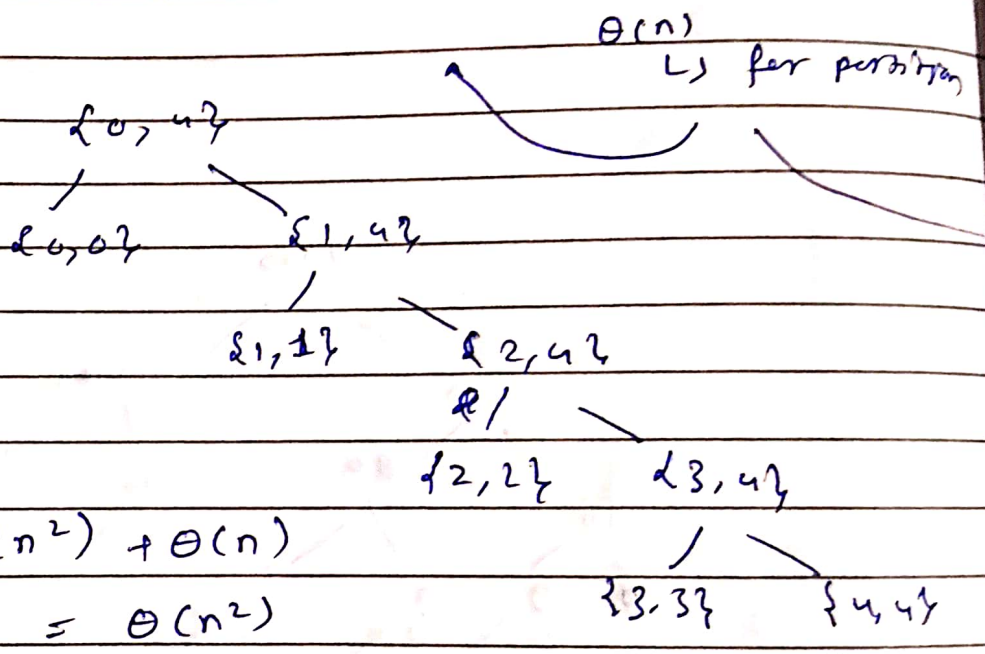
= θ(n log n)

Scanned with CamScanner

Worst case



Worst case: → $T(n) = T(n-1) + \Theta(n)$



$\Theta(n^2) + \Theta(n)$

$= \Theta(n^2)$

Scanned with CamScanner

=> Space Analysis

Hoare, Lomuto → both inplace

but in quick sort → In-place is arguable

↓

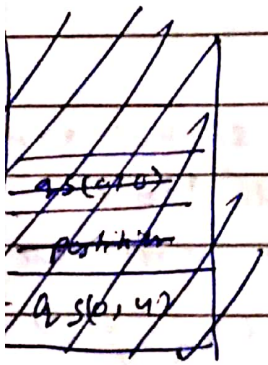
It depends on how we define quick sort.

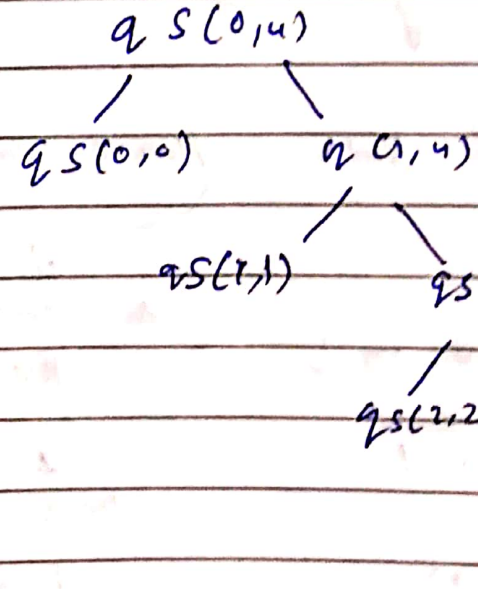
quick sort

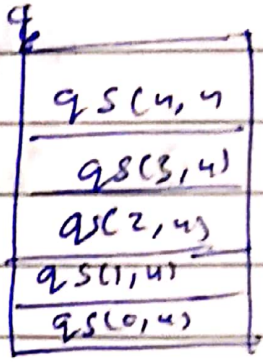
↙ constant space → Then not In-place bcz of recursion it will take extra space in ~~stack~~ Queue Stack

↘ does not copy to Auxillary space → In-place

Worst Case







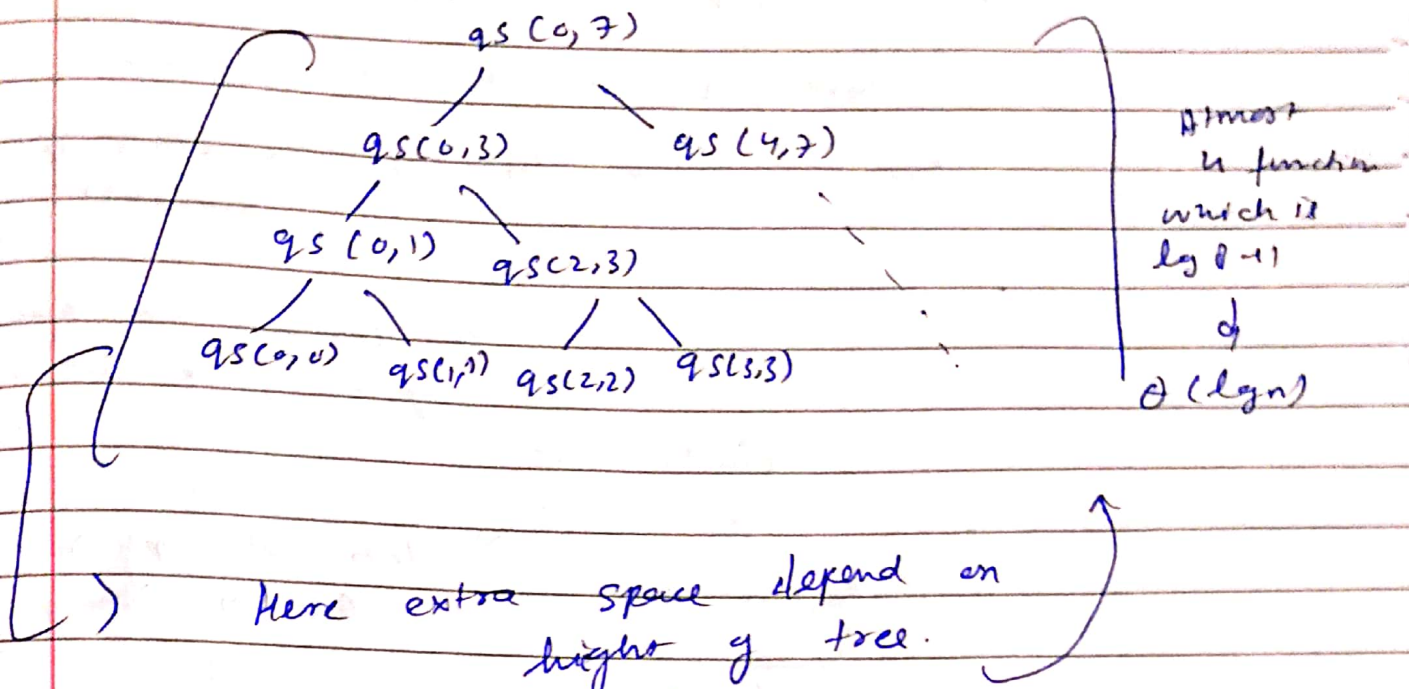
In worst case

↓

$\Theta(n)$

Scanned with CamScanner

Best case :



Here extra space depend on height of tree.

↓

it will become worst case ⇒ $\Theta(n)$ when we divide $(n-1)$

Best ↳ half - half division

⇒ Choice of Pivot and worst case of Quick sort

↳ Both Hoare and Lomuto

↳ in case of sorted array they will lead into worst case

↳ bcz of div of 1 & (n-1)

↳ Random function to generate pivot

{10, 50, 100, 200}

1 — n-1

p = random (l, r)

swap (arr[l], arr[p])

↳ Hoare's

↳ swap (arr[l], arr[p])

Lomuto

Scanned with CamScanner